Geant4 User's Guide - For Toolkit Developers

June 28, 2005

Contents

I Introduction 1

1 Introduction 2

1.1 Scope of this manual 2
1.2 How to use this manual 2
1.3 User Requirements Document 3
1.4 Status of this chapter 3

II Design and Function of Geant4 Categories 4

2 Introduction 5

3 Run 6

3.1 Design Philosophy 6
3.2 Class Design 6
3.3 Status of this chapter 6

4 Event 7

4.1 Design Philosophy 7
4.2 Class Design 7
4.3 Status of this chapter 8

5 Tracking 11

5.1 Design Philosophy 11
5.2 Class Design 12
5.3 Tracking Algorithm 13
5.4 Interaction with Physics Processes 15
5.5 Ordering of Methods of Physics Processes 18
5.6 Status of this chapter 19

6 Physics Processes **21**
6.1 Design Philosophy 21
6.2 Class Design 22
6.2.1 General 22
6.3 Status of this chapter 22

7 Hits and Digitization **25**
7.1 Design Philosophy 25
7.2 Class Design 25
7.3 Status of this chapter 28

8 Geometry **29**
8.1 Design Philosopy 29
8.2 Class Design 30
8.3 Status of this chapter 33

9 Electromagnetic Fields **35**

10 Particles **37**
10.1 Design Philosophy 37
10.2 Class Design 37
10.3 Status of this chapter 38

11 Materials **40**
11.1 Design Philosophy 40
11.2 Class Design 40
11.3 Status of this chapter 40

12 Global Usage **42**
12.1 Design Philosophy 42
12.2 Class Design 43
12.3 Status of this chapter 47

13 Visualization **48**
13.1 Design Philosophy 48
13.2 Class Design 49
13.3 Modeling sub-category 50
13.4 View parameters 51
13.5 Status of this section 51

14 Intercoms **52**
14.1 Design Philosophy 52
14.2 Class Design 52
14.3 Status of this section 53

III Extending Toolkit Functionality **54**

15 Geometry **55**
15.1 What can be extended ? 55
15.2 Adding a new type of Solid 56
15.3 Modifying the Navigator 60

16 Electromagnetic Fields **61**
16.1 Creating a New Type of Field 61
16.2 Status of this chapter 64

17 Physics Processes **65**
17.1 Status of this chapter 65

18 Hadronic Physics **66**
18.1 Introduction 66
18.2 Principal Considerations 66
18.3 Level 1 Framework - processes 67
18.4 Level 2 Framework - Cross Sections and Models 69
18.5 Level 3 Framework - Theoretical Models 74
18.6 Level 4 Frameworks - String Parton Models and Intra-Nuclear Cascade 77
18.7 Level 5 Framework - String De-excitation 79

19 Visualization **82**

IV Appendix **83**

20 Class Diagrams for the Geometry Category **84**

Part I

Introduction

Chapter 1

Introduction

1.1 Scope of this manual

The User's Guide for Toolkit Developers provides detailed information about the design of Geant4 classes as well as the information required to extend the current functionality of the Geant4 toolkit. This manual is designed to:

- provide a repository of information for those who want to understand or refer to the detailed design of the toolkit, and
- provide details and procedures for extending the functionality of the toolkit so that experienced users may contribute code which is consistent with the overall design of Geant4.

This manual is intended for developers and experienced users of Geant4. It is assumed that the reader is already familiar with functionality of the Geant4 toolkit as explained in the "User's Guide For Application Developers", and also has a working knowledge of programming using C++. A knowledge of object-oriented analysis and design will also be useful in understanding this manual. It is also useful to consult the "Software Reference Manual" which provides a list of Geant4 classes and their major methods.

Detailed discussions of the physics included in Geant4 are provided in the "Physics Reference Manual".

1.2 How to use this manual

To understand the goal of the software design of Geant4, it is useful to begin by reading the User Requirements Document referred to in **this chapter**.

Chapter 2, "Design and Function of the Geant4 Categories" provides detailed information about the design of each class category and the classes in it. Before considering an extension of one of the toolkit categories, a detailed understanding of that category is required.

Chapter 3, "Extending Toolkit Functionality" explains in some detail how to extend the functionality of Geant4. Most of the class categories are covered and some, which are especially useful to most users, are covered in greater detail.

It is not necessary to understand the entire manual before adding a new functionality. To add a new physics process, for example, only the following items must be read and understood:

- the design principle described in the "Physics processes" section of Chapter 2
- techniques explained in the "Physics processes" section of Chapter 3.

1.3 User Requirements Document

At the beginning of Geant4 development, a set of user requirements was collected in order to inform the object-oriented analysis and design of the toolkit. The User Requirements Document follows the PSS-05 software engineering standards and is available at

`http://cern.ch/geant4/OOAandD/URD.pdf` .

This document provides a general description of the main capabilities and constraints of the toolkit. It also defines three types of users characterized by their level of interaction with the system. Specific requirements are also listed and classified.

1.4 Status of this chapter

24.06.05 - re-organized and re-written by D.H. Wright

Part II

Design and Function of Geant4 Categories

Chapter 2

Introduction

Geant4 exploits advanced software engineering techniques based on the Booch/UML Object Oriented Methodology and follows the evolution of the ESA Software Engineering Standards for the development process. The ”spiral”, or iterative, approach has been adopted. User requirements were collected in the initial phase and problem domain decomposition, object-oriented methods, and CASE tools were used for analysis and design. This produced a clear hierarchical structure of sub-domains linked by a uni-directional flow of dependencies. This led to a software product which is modular and flexible (a toolkit) and in which the physics implementation is transparent and open to user validation of physics predictions. It allows the user to understand, customize and extend the toolkit in all domains. At the same time the modular architecture allows the user to load only needed components.

Chapter 3

Run

3.1 Design Philosophy

The run category manages collections of events that share a common beam and detector implementation.

3.2 Class Design

- **G4Run** - This class represents a run. An object of this class is constructed and deleted by G4RunManager.
- **G4RunManager** - the run controller class. Users must register detector construction, physics list and primary generator action classes to it. G4RunManager or a derived class must be a singleton.
- **G4RunManagerKernel** - provides control of the Geant4 kernel. This class is constructed by G4RunManager.

3.3 Status of this chapter

28.06.05 under construction

Chapter 4

Event

4.1 Design Philosophy

In high energy physics the primary unit of an experimental run is an event. An event consists of a set of primary particles produced in an interaction, and a set of detector responses to these particles.

In Geant4, objects of the *G4Event* class are the primary units of a simulation run. Before the event is processed, it contains primary vertices and primary particles produced by an external physics generator. After the event is processed, it may also contain hits, digitizations ,and optionally, trajectories generated by the simulation. The event category manages events and provides an abstract interface to external physics generators.

G4Event and its content vertices and particles are independent of other classes. This isolation allows Geant4-based simulation programs to be independent of specific choices for physics generators and of specific solutions for storing the "Monte Carlo truth". *G4Event* avoids keeping any transient information which is not meaningful after event processing is complete. Thus the user can store objects of this class for processing further down the program chain. For performance reasons, *G4Event* and its content classes are not persistent. Instead the user must provide the transient-to-persistent conversion.

4.2 Class Design

- **G4Event** - This class represents an event. It is constructed and deleted by G4RunManager or its derived class.
- **G4EventManager** - This class controls an event. It must be a sin-

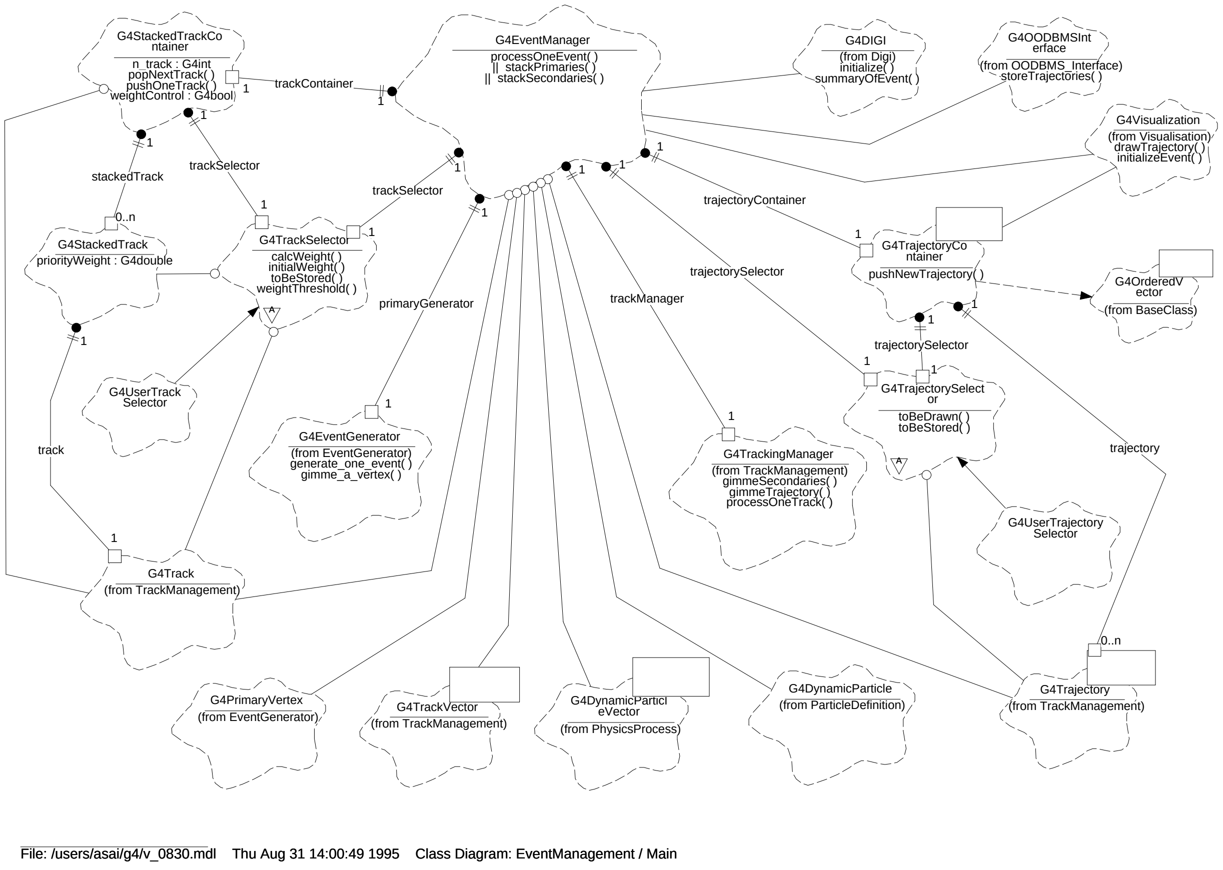


Figure 4.1: Event

gleton and should be constructed by G4RunManager.

- **G4VPrimaryGenerator** - the abstract base class of all of primary generators. This class has only one pure virtual method, GeneratePrimaryVertex(), which takes a G4Event object, generates a primary vertex and associates primary particles with the vertex.

Booch diagrams for classes related to the event and event generator classes are shown in Figs. 4.1 and 4.2, respectively.

4.3 Status of this chapter

27.06.05 design philosophy section added (from Geant4 general paper) by D.H. Wright

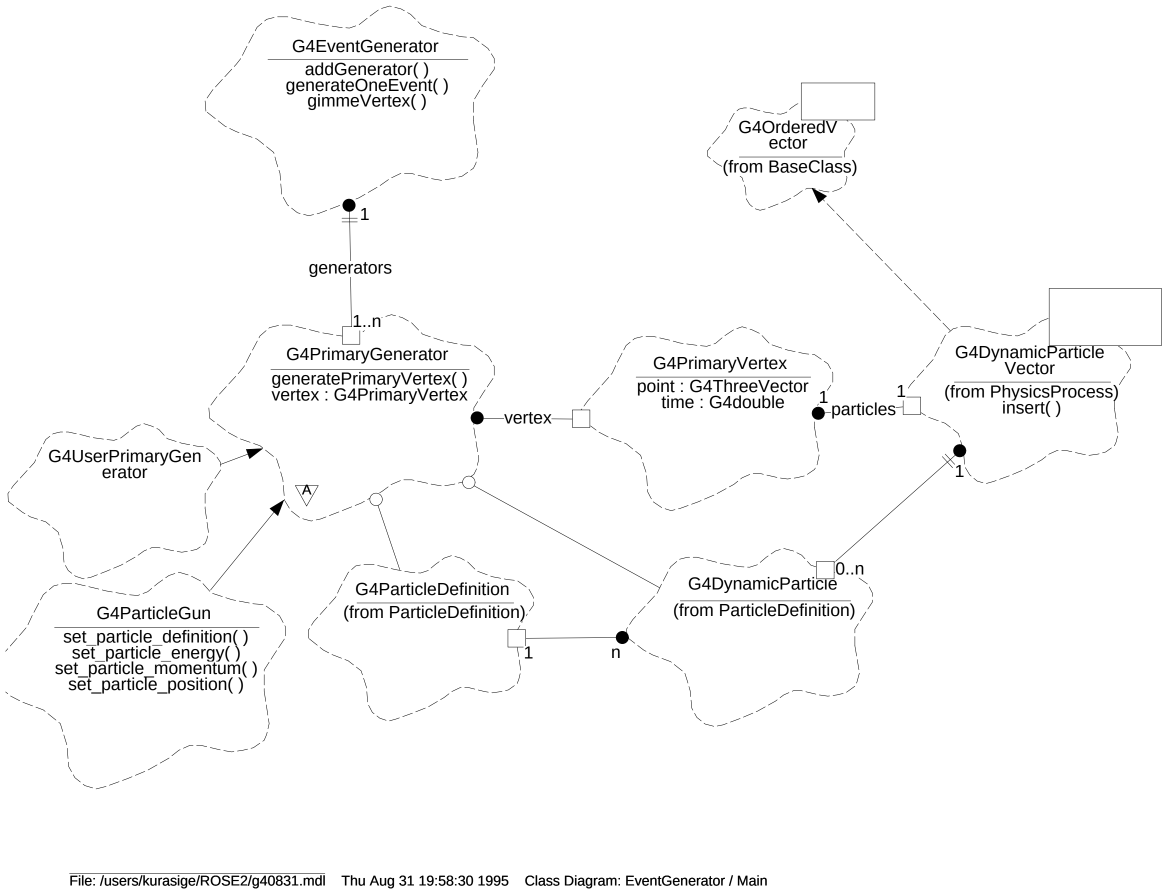


Figure 4.2: Event Generator

Chapter 5

Tracking

The tracking category manages the contribution of the processes to the evolution of a track's state and provides information in sensitive volumes for hits and digitization.

5.1 Design Philosophy

It is well known that the overall performance of a detector simulation depends critically on the CPU time spent propagating the particle through one step. The most important consideration in the object design of the tracking category is maintaining high execution speed in the Geant4 simulation while utilizing the power of the object-oriented approach.

An extreme approach to the particle tracking design would be to integrate all functionalities required for the propagation of a particle into a single class. This design approach looks object-oriented because a particle in the real world propagates by itself while interacting with the material surrounding it. However, in terms of data hiding, which is one of the most important ingredients in the object-oriented approach, the design can be improved.

Combining all the necessary functionalities into a single class exposes all the data attributes to a large number of methods in the class. This is basically equivalent to using a common block in Fortran.

Instead of the 'big-class' approach, a hierarchical design was employed by Geant4. The hierarchical approach, which includes inheritance and aggregation, enables large, complex software systems to be designed in a structured way. The simulation of a particle passing through matter is a complex task involving particles, detector geometry, physics interactions and hits in the detector. It is well-suited to the hierarchical approach. The hierarchical design manages the complexity of the tracking category by separating the system into layers. Each layer may then be designed independently of the others.

In order to maintain high-performance tracking, use of the inheritance ('is-a' relation) hierarchy in the tracking category was avoided as much as possible. For example, *track* and *particle* classes might have been designed so that a *track* 'is a' *particle*. In this scheme, however, whenever a *track* object is used, time is spent copying the data from the *particle* object into the *track* object. Adopting the aggregation ('has-a' relation) hierarchy requires only pointers to be copied, thus providing a performance advantage.

5.2 Class Design

Fig. (not yet available) shows a general overview of the tracking design in Unified Modelling Language Notation.

- *G4TrackingManager* is an interface between the event and track categories and the tracking category. It handles the message passing between the upper hierarchical object, which is the event manager (`G4EventManager`), and lower hierarchical objects in the tracking category. `G4TrackingManager` is responsible for processing one track which it receives from the event manager.

  `G4TrackingManager` aggregates the pointers to `G4SteppingManager`, `G4Trajectory` and `G4UserTrackingAction`. It also has a 'use' relation to `G4Track`.

- *G4SteppingManager* plays an essential role in particle tracking. It performs message passing to objects in all categories related to particle transport, such as geometry and physics processes. Its public method `Stepping()` steers the stepping of the particle. The algorithm employed in this method is basically the same as that in Geant3. The GEANT4 implementation, however, relies on the inheritance hierarchy of the physics interactions. The hierarchical design of the physics interactions enables the stepping manager to handle them as abstract objects. Hence, the manager is not concerned with concrete interaction objects such as bremsstrahlung or pair creation. The actual invocations of various interactions during the stepping are done through a dynamic binding mechanism. This mechanism shields the tracking category from any change in the design of the physics process classes, including the addition or subtraction of new processes.

  `G4SteppingManager` also aggregates

    - the pointers to `G4Navigator` from the geometry category, to the current `G4Track`, and
    - the list of secondaries from the current track (through a `G4TrackVector`) to `G4UserSteppingAction` and to `G4VSteppingVerbose`.

  It also has a 'use' relation to `G4ProcessManager` and `G4ParticleChange` in the physics processes class category.

- *G4Track* - the class `G4Track` represents a particle which is pushed by `G4SteppingManager`. It holds information required for stepping a particle, for example, the current position, the time since the start of stepping, the identification of the geometrical volume which contains the particle, etc. Dynamic information, such as particle momentum and energy, is held in the class through a pointer to the `G4DynamicParticle` class. Static information, such as the particle mass and charge is stored in the `G4DynamicParticle` class through the pointer to the `G4ParticleDefinition` class. Here the aggregation hierarchical design is extensively employed to maintain high tracking performance.

- *G4TrajectoryPoint* and *G4Trajectory* - the class `G4TrajectoryPoint` holds the state of the particle after propagating one step. Among other things, it includes information on space-time, energy-momentum and geometrical volumes.

  `G4Trajectory` aggregates all `G4TrajectoryPoint` objects which belong to the particle being propagated. `G4TrackingManager` takes care of adding the `G4TrajectoryPoint` to a `G4Trajectory` object if the user requested it (see [1]). The life of a `G4Trajectory` object spans an event, contrary to `G4Track` objects, which are deleted from memory after being processed.

- G4UserTrackingAction and G4UserSteppingAction - `G4UserTrackingAction` is a base class from which user actions at the beginning or end of tracking may be derived. Similarly, `G4UserSteppingAction` is a base class from which user actions at the beginning or end of each step may be derived.

5.3 Tracking Algorithm

The key classes for tracking in GEANT4 are `G4TrackingManager` and `G4SteppingManager`. The singleton object "TrackingManager" from `G4TrackingManager` keeps all information related to a particular track, and it also manages all actions necessary to complete the tracking. The tracking proceeds by pushing a particle by a step, the length of which is defined by one of the active processes. The "TrackingManager" object delegates management of each of the steps to the "SteppingManager" object. This object keeps all information related to a particular step.

The public method `ProcessOneTrack()` in `G4TrackingManager` is the key to managing the tracking, while the public method `Stepping()` is the key to managing one step. The algorithms used in these methods are explained below.

ProcessOneTrack() in G4TrackingManager

1. Actions before tracking the particle:
   Clear secondary particle vector
2. Pre tracking user intervention process.
3. Construct a trajectory if it is requested
4. Give SteppingManager the pointer to the track which will be tracked
5. Inform beginning of tracking to physics processes
6. Track the particle Step-by-Step while it is alive
   - Call Stepping method of G4SteppingManager
   - Append a trajectory point to the trajectory object if it is requested
7. Post tracking user intervention process.
8. Destroy the trajectory if it was created

Stepping() in G4SteppingManager

1. Initialize current step
2. If particle is stopped, get the minimum life time from all the at rest processes and invoke InvokeAtRestDoItProcs for the selected AtRest processes
3. If particle is not stopped:
   - Invoke DefinePhysicalStepLength, that finds the minimum step length demanded by the active processes

- Invoke InvokeAlongStepDoItProcs
- Update current track properties by taking into account all changes by AlongStepDoIt
- Update the *safety*
- Invoke PostStepDoIt of the active discrete process.
- Update the track length
- Send G4Step information to Hit/Dig if the volume is sensitive
- Invoke the user intervention process.
- Return the value of the StepStatus.

5.4 Interaction with Physics Processes

The interaction of the tracking category with the physics processes is done in two ways. First each process can limit the step length through one of its three `GetPhysicalInteractionLength()` methods, AtRest, AlongStep, or PostStep. Second, for the selected processes the DoIt (AtRest, AlongStep or PostStep) methods are invoked. All this interaction is managed by the Stepping method of `G4SteppingManager`. To calculate the step length, the `DefinePhysicalStepLength()` method is called. The flow of this method is the following:

- Obtain maximum allowed Step in the volume define by the user through G4UserLimits.

- The PostStepGetPhysicalInteractionLength of all active processes is called. Each process returns a step length and the minimum one is chosen. This method also returns a G4ForceCondition flag, to indicate if the process is forced or not: = Forced : Corresponding PostStepDoIt is forced. = NotForced : Corresponding PostStepDoIt is not forced unless this process limits the step. = Conditionally : Only when AlongStepDoIt limits the step, corresponding PoststepDoIt is invoked. = ExclusivelyForced : Corresponding PostStepDoIt is exclusively forced. All other DoIt including AlongStepDoIts are ignored.

- The AlongStepGetPhysicalInteractionLength method of all active processes is called. Each process returns a step length and the minimum of these and the This method also returns a fGPILSelection flag, to indicate if the process is the selected one can be is forced or not: = CandidateForSelection: this process can be the winner. If its step length is the smallest, it will be the process defining the step (the process = NotCandidateForSelection: this process cannot be the winner. Even if its step length is taken as the smallest, it will not be the process defining the step

The method `G4SteppingManager::InvokeAlongStepDoIts()` is in charge of calling the AlongStepDoIt methods of the different processes:

- If the current step is defined by a 'ExclusivelyForced' PostStepGetPhysicalInteractionLength, no AlongStepDoIt method will be invoked

- Else, all the active continuous processes will be invoked, and they return the ParticleChange. After it for each process the following is executed:

  - Update the G4Step information by using final state information of the track given by a physics process. This is done through the UpdateStepForAlongStep method of the ParticleChange
  - Then for each secondary:
    - It is checked if its kinetic energy is smaller than the energy threshold for the material. In this case the particle is assigned a 0. kinetic energy and its energy is added as deposited energy of the parent track. This check is only done if the flag ApplyCutFlag is set for the particle (by default it is set to 'false' for all particles, user may change it in its G4VUserPhysicsList). If the track has the flag IsGoodForTracking 'true' this check will have no effect (used mainly to track particles below threshold)
    - The parentID and the process pointer which created this track are set
    - The secondary track is added to the list of secondaries. If it has 0. kinetic energy, it is only added if it it invokes a rest process at the beginning of the tracking
  - The track status is set according to what the process defined

The method G4SteppingManager::InvokePostStepDoIts is on charge of calling the PostStepDoIt methods of the different processes.

- Invoke the PostStepDoIt methods of the specified discrete process (the one selected by the PostStepGetPhysicalInteractionLength, and they return the ParticleChange. The order of invocation of processes is inverse to the order used for the GPIL methods. After it for each process the following is executed:

  - Update PostStepPoint of Step according to ParticleChange
  - Update G4Track according to ParticleChange after each PostStepDoIt
  - Update safety[1] after each invocation of PostStepDoIts
  - The secondaries from ParticleChange are stored to SecondaryList
  - Then for each secondary:
    * It is checked if its kinetic energy is smaller than the energy threshold for the material. In this case the particle is assigned a 0. kinetic energy and its energy is added as deposited energy of the parent track. This check is only done if the flag ApplyCutFlag is set for the particle (by default it is set to 'false' for all particles, user may change it in its G4VUserPhysicsList). If the track has the flag IsGoodForTracking 'true' this check will have no effect (used mainly to track particles below threshold)
    * The parentID and the process pointer which created this track are set
    * The secondary track is added to the list of secondaries. If it has 0. kinetic energy, it is only added if it it invokes a rest process at the beginning of the tracking
  - The track status is set according to what the process defined

The method G4SteppingManager::InvokeAtRestDoIts is called instead of the three above methods in case the track status is *fStopAndALive*. It is on charge of selecting the rest process which has the shortest time before and then invoke it:

- To select the process with shortest tiem, the AtRestGPIL method of all active processes is called. Each process returns an lifetime and the minimum one is chosen. This method returm also a G4ForceCondition flag, to indicate if the process is forced or not: = Forced : Corresponding AtRestDoIt is forced. = NotForced : Corresponding AtRestDoIt is not forced unless this process limits the step.

- Set the step length of current track and step to 0.

- Invoke the AtRestDoIt methods of the specified at rest process, and they return the ParticleChange. The order of invocation of processes is inverse to the order used for the GPIL methods.

[1]

After it for each process the following is executed:

- Set the current process as a process which defined this Step length.
- Update the G4Step information by using final state information of the track given by a physics process. This is done through the UpdateStepForAtRest method of the ParticleChange.
- The secondaries from ParticleChange are stored to SecondaryList
- Then for each secondary:
    - It is checked if its kinetic energy is smaller than the energy threshold for the material. In this case the particle is assigned a 0. kinetic energy and its energy is added as deposited energy of the parent track. This check is only done if the flag ApplyCutFlag is set for the particle (by default it is set to 'false' for all particles, user may change it in its G4VUserPhysicsList). If the track has the flag IsGoodForTracking 'true' this check will have no effect (used mainly to track particles below threshold)
    - The parentID and the process pointer which created this track are set
    - The secondary track is added to the list of secondaries. If it has 0. kinetic energy, it is only added if it it invokes a rest process at the beginning of the tracking
- The track is updated and its status is set according to what the process defined

5.5 Ordering of Methods of Physics Processes

The ProcessManager of a particle is responsible for providing the correct ordering of process invocations. `G4SteppingManager` invokes the processes at each phase just following the order given by the ProcessManager of the corresponding particle.

For some processes the order is important. GEANT4 provides by default the right ordering. It is always possible for the user to choose the order of process invocations at the initial set up phase of GEANT4. This default ordering is the following:

1. Ordering of GetPhysicalInteractionLength

    - In the loop of GetPhysicalInteractionLength of AlongStepDoIt, the Transportation process has to be invoked at the end.

- In the loop of GetPhysicalInteractionLength of AlongStepDoIt, the Multiple Scattering process has to be invoked just before the Transportation process.

2. Ordering of DoIts

    - There is only some special cases. For example, the Cherenkov process needs the energy loss information of the current step for its DoIt invocation. Therefore, the EnergyLoss process has to be invoked before the Cherenkov process. This ordering is provided by the process manager. Energy loss information necessary for the Cherenkov process is passed using G4Step (or the static dE/dX table is used together with the step length information in G4Step to obtain the energy loss information). Any other?

5.6 Status of this chapter

created by ?
10.06.02 partially re-written by D.H. Wright
14.11.02 updated and partially re-written by P. Arce

Bibliography

[1] Geant4 Users Guide for Application Developers.

Chapter 6

Physics Processes

6.1 Design Philosophy

The processes category contains the implementations of particle transportation and physical interactions. All physics process conform to the basic interface *G4VProcess*, but different approaches have been developed for the detailed design of each sub-category.

For the decay sub-category, the decays of all long-lived, unstable particles are handled by a single process. This process gets the step length from the mean life of the particle. The generation of decay products requires a knowledge of the branching ratios and/or data distributions stored in the particle class.

The electromagnetic sub-category is divided further into the following packages:

- *standard*: handling basic properties for electron, positron, photon and hadron interactions,
- *low energy*: providing alternative models extended down to lower energies than the standard package,
- *muons*: handling muon interactions,
- *x-rays*: providing specific code for x-ray physics,
- *optical*: providing specific code for optical photons,
- *utils*: collecting utility classes used by the above packages.

It provides the features of openness and extensibilty resulting from the use of object-oriented technology; alternative physics models, obeying the same process abstract interface, are often available for a given type of interaction.

For hadronic physics, an additional set of implementation frameworks was added to accommodate the large number of possible modeling approaches. The top-level framework provides the basic interface to other GEANT4 categories. It satisfies the most general use-case for hadronic shower simulations, namely to provide inclusive cross sections and final state generation. The frameworks are then refined for increasingly specific use-cases, building a hierarchy in which each level implements the interface specified by the level above it. A given hadronic process may be implemented at any one of these levels. For example, the process may be implemented by one of several models, and each of the models may in turn be implemented by several sub-models at the lower framework levels.

6.2 Class Design

6.2.1 General

The object-oriented design of the generic physics process G4VProcess and its relation to the process manager is shown in Fig. 6.1. Fig. 6.2 shows how specific physics processes are related to G4VProcess.

6.3 Status of this chapter

27.06.05 section on design philosophy added by D.H. Wright

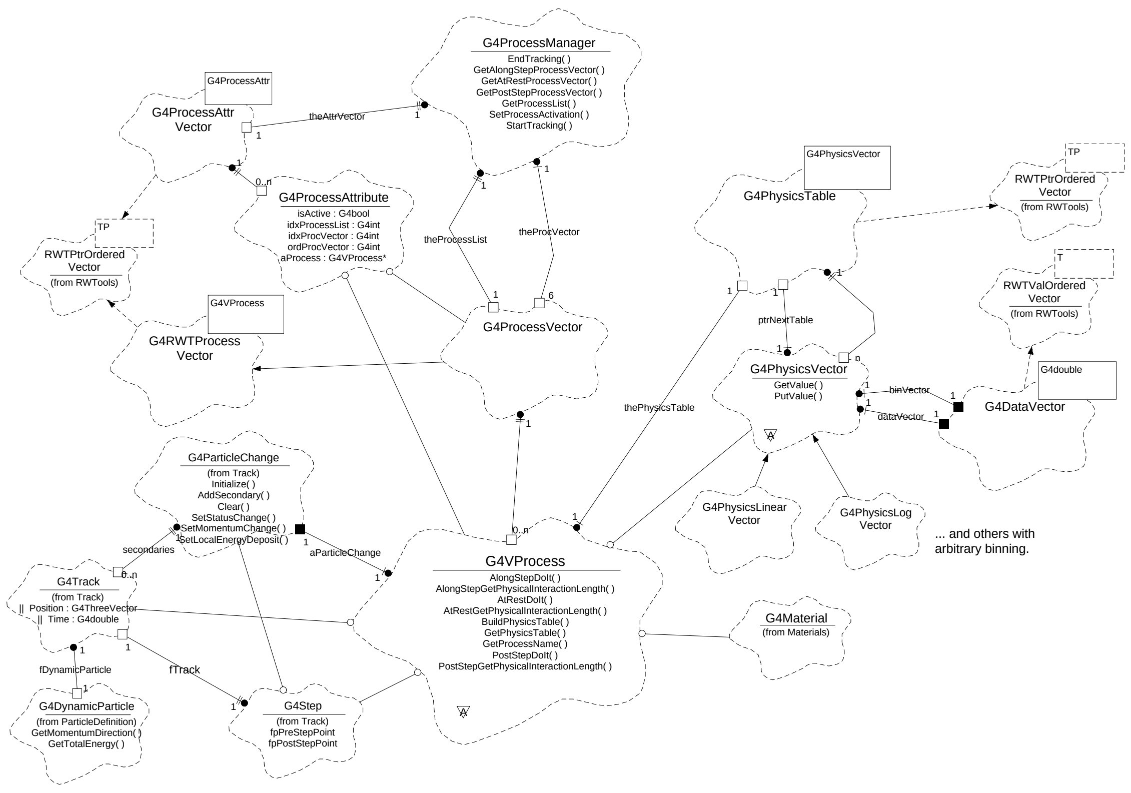


Figure 6.1: Management of Physics Processes

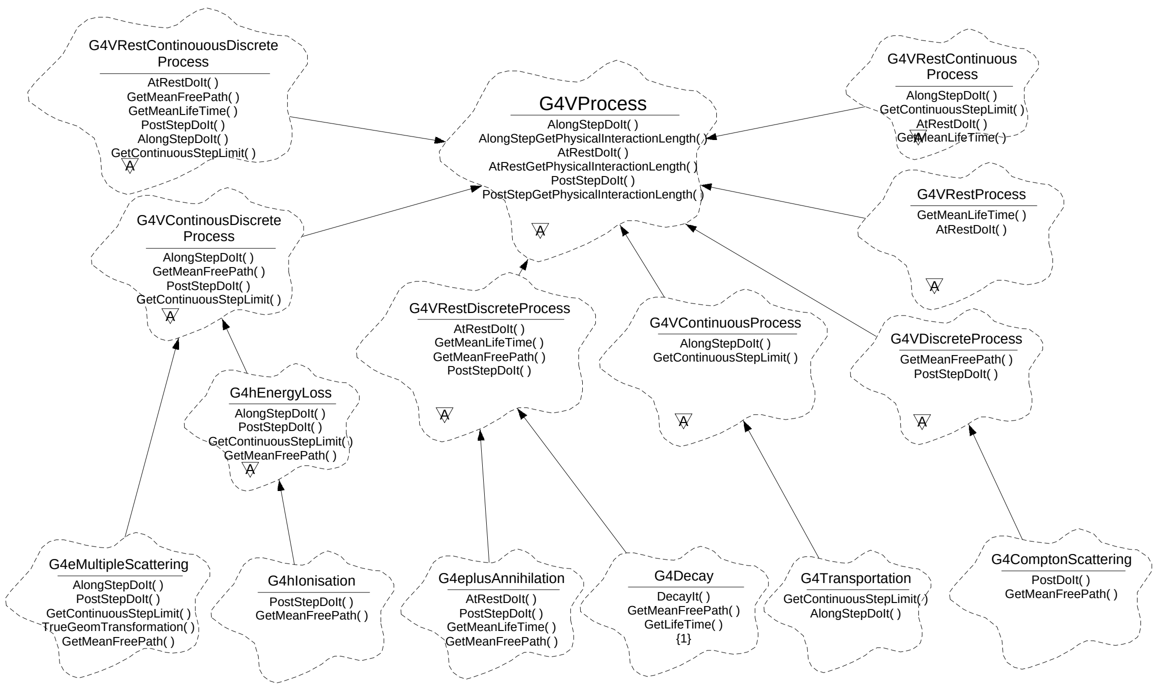


Figure 6.2: General Physics Processes

Chapter 7

Hits and Digitization

7.1 Design Philosophy

In GEANT4 a *hit* is a snapshot of a physical interaction or an accumulation of interactions of a track or tracks in a "sensitive" detector component. A digitization, or *digit*, represents a detector output, such as an ADC/TDC count or a trigger signal. A *digit* is created from one or more hits and/or other *digits*. Given the wide variety of GEANT4 applications, ways of describing detector sensitivity and the quantities to be stored in the *hits* and *digits* vary greatly. This category therefore provides only abstract classes for both detector sensitivity and *hits/digits*. It also provides tools for organizing the *hits/digits* into collections.

7.2 Class Design

- **G4SensitiveDetectorManager** - a list of G4SensitiveDetectors.
- **G4HitsStructure** - a tree-like structure of G4Hit collections. Each branch represents the hits in given sub-detector. For example, the first level of branches may consist of a tracker, ECAL, and HCAL, while the second level, in HCAL, consists of the barrel and endcaps. Finally the barrel may have phi-slices, Z-slices, etc.
- **G4VSensitiveDetector** - an abstract class of all of sensitive volumes.
- **G4HitsCollection** - a collection of hits. Instantiates an RWCollection class.
- **G4VHit** - this class has all the information about a particular hit caused by a single step.

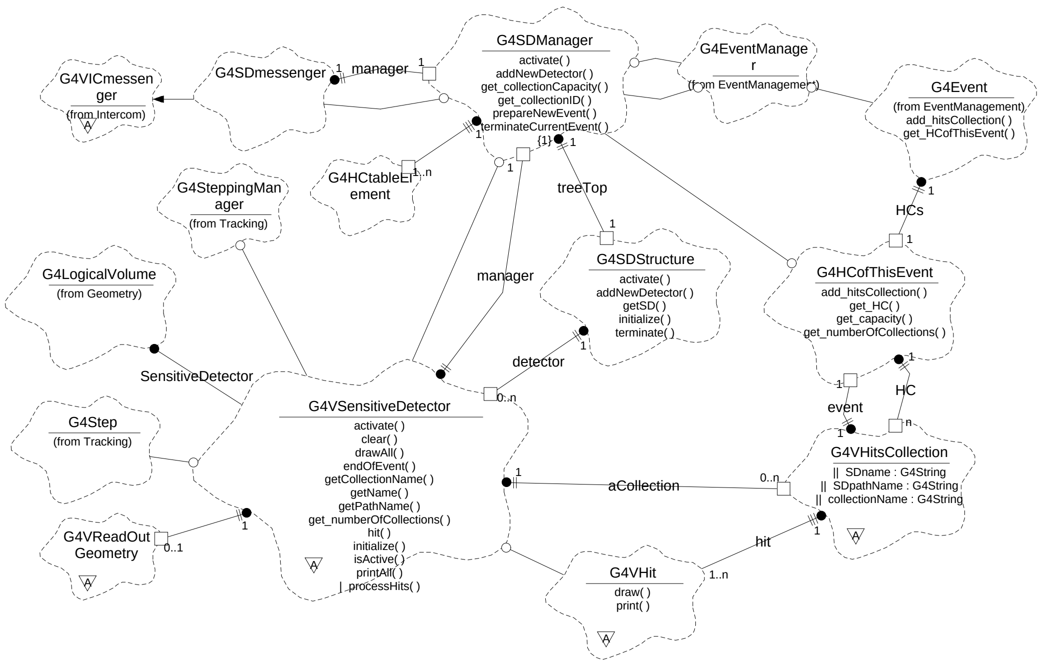


Figure 7.1: Overview of hit classes management

- **G4VDigitizer** - the class of objects which transform the hits deposited by particles into digitizations.
- **G4DigitizerManager** - the (single) object dispatching common messages to individual digitizers.
- **G4VDigi** - an abstract (base) class for all G4 digitizations. This could be data as simple as a singe byte, or as complex as an Ntuple.
- **G4DigiStructure** - digitizations are organized as a structure, which could be anything between a single value and an Ntuple.

The object-oriented design of the 'hit' related classes is shown in the following class diagrams. The diagrams are described in the Booch notation. Fig. 7.1 shows the general management of hit classes. Fig. 7.2 shows the OO design of user-related hit classes. Fig. 7.3 shows the OO design of the readout geometry.

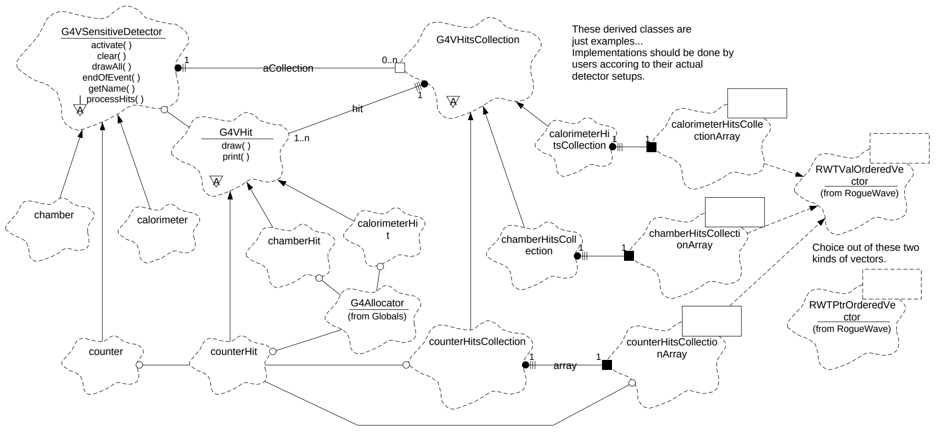


Figure 7.2: User hit classes

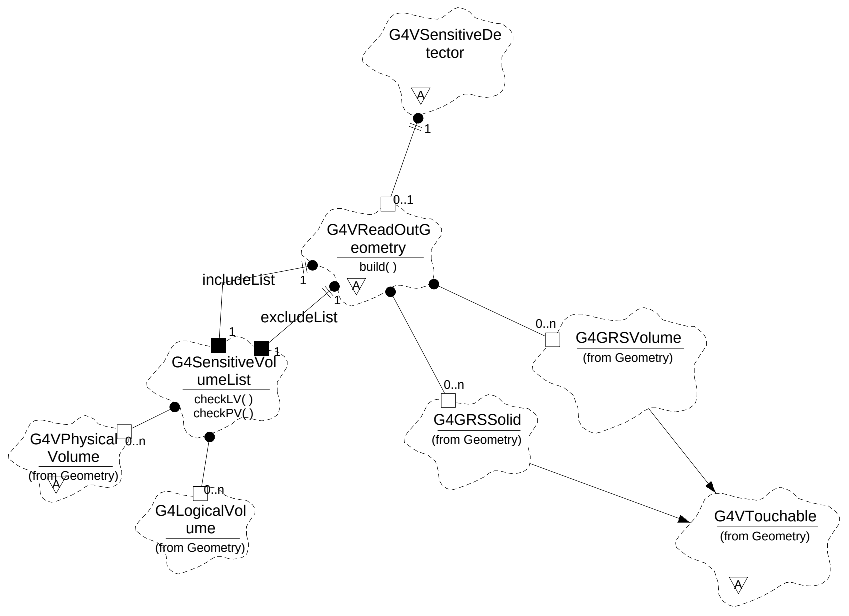


Figure 7.3: Readout geometry

7.3 Status of this chapter

27.06.05 section on design philosophy added (from Geant4 general paper) by D.H. Wright

Chapter 8

Geometry

8.1 Design Philosopy

The geometry category provides the ability to describe a geometrical structure and propagate particles efficiently through it. This is done in part with the aid of two central concepts, the *logical* and *physical* volumes. A logical volume represents a detector element of a given shape which may contain other volumes, and which may have other attributes. It has access to other information which is independent of its phyisical location in the detector, such as material and sensitive detector behavior. A physical volume represents the spatial positioning or placement of the logical volume with respect to an enclosing mother (logical) volume. Thus a hierarchical tree structure of volumes can be built with each volume containing smaller volumes (which may not overlap). Repetitive structures can be represented by specialized physical volumes, such as replicas and parameterized placements, sometimes resulting in a large savings in memory.

In GEANT4 the logical volume has been refined by defining the shape as a separate entity, called a *solid.* Solids with simple shapes, like rectilinear boxes, trapezoids, spherical or cylindrical sections or shells, each have their properties coded separately, in accord with the concept of *Constructed Solid Geometry (CSG).* More complex solids are defined by their bounding surfaces, which can be planes, second-order surfaces or higher-order B-spline surfaces, and belong to the *Boundary Representations (BREP)* sub-category.

Another way to build solids is by boolean combination - union, intersection and subtraction. The elemental solids should be CSGs.

Although a detector is naturally and best described as by a hierarchy of volumes, efficiency is not critically dependent on this. An optimization technique, called voxelization, allows efficient navigation even in "flat" geometries, typical of those produced by CAD systems.

8.2 Class Design

- **G4GeometryManager** - responsible for managing "high level" objects in the geometry subdomain, notably including opening and closing ("locking") the geometry, and creating/deleting optimization information for G4Navigator. The class is a "singleton".

- **G4LogicalVolumeStore** - a container for optionally storing created logical volumes. It enables traversal of all logical volumes by the UI/user/etc.

- **G4LogicalVolume** - represents a leaf node or unpositioned subtree in the geometry hierarchy. It may have daughters ascribed to it, and is also responsible for retrieval of the physical and tracking attributes of the physical volume that it represents. These attributes include solid, material, magnetic field, and optionally user limits, sensitive detectors, etc. Logical volumes are optionally entered into the G4LogicalVolumeStore.

- **G4MagneticField** - a class responsible for the magnetic field in each volume, including the calculation of particle trajectories along curved paths. In cases where the geometry step limits the particle's step, the distance calculated is guaranteed to be the distance to a volume boundary.

- **G4Navigator** - a class used by the tracking management, able to obtain/calculate tracking-time geometrical information such as distance to the next volume, or to find the physical volume containing a given point in the world reference system. The navigator maintains a transformation history and other information used to optimize the tracking time performance.

- **G4NavigationHistory** - responsible for maintenance of the history of the path taken through the geometrical hierarchy. It is principally a utility class for use by G4Navigator.

- **G4NormalNavigation** - a utility class for navigation in volumes containing only G4PVPlacement daughter volumes.

- **G4ParameterisedNavigation** - a utility class for navigation in volumes containing a single G4PVParameterised volume for which voxels for the replicated volumes have been constructed.

- **G4VoxelNavigation** - a utility class for navigation in volumes containing only G4PVPlacement daughter volumes for which voxels have been constructed.
- **G4ReplicaNavigation** - a utility class for navigation in volumes containing a single G4PVParameterised volume for which voxels for the replicated volumes have been constructed.
- **G4PhysicalVolumeStore** - a container for optionally storing created physical volumes. It enables traversal of all physical volumes by the UI/user/etc. All solids should be registered with G4PhysicalVolumeStore, and removed on their destruction. It is intended principally for the UI browser.
- **G4VPhysicalVolume** - a volume positioned within and relative to a given mother volume, and also represented by a given logical volume. They are optionally entered into the G4PhysicalVolumeStore.
- **G4PVPlacement** - a physical volume corresponding to a single touchable detector element, positioned within and relative to a mother volume.
- **G4PVIndexed** - a volume able to perform simple changes to its shape (corresponds to GSPOSP), and representing a single touchable detector element.
- **G4PVReplica** - a physical volume representing many identically formed touchable detector elements, differing only in their positioning. The elements' positions are determined by means of a simple formula, and the elements completely fill the containing mother volume.
- **G4PVParameterised** - a physical volume representing many touchable detector elements differing in their positioning and dimensions. Both are calculated by means of a G4VParameterisation object. Each element's position is calculated as per G4PVReplica, and each element's shape can be modified by means of a user supplied formula.
- **G4VPVParameterisation** - a parameterisation class able to compute the transformation and, indirectly, the dimensions of parameterised volumes, given a replication number.
- **G4SmartVoxelProxy** - a class for proxying smart voxels. The class represents either a header (in turn refering to more VoxelProxies) or a

node. If created as a node, calls to GetHeader cause an exception, and likewise GetNode when a header.

- **G4SmartVoxelHeader** - represents a single axis of virtual division. Contains the individual divisions which are potentially further divided along different axes.

- **G4SmartVoxelNode** - a single virtual division, containing the physical volumes inside its boundaries and those of its parents.

- **G4VoxelLimits** - represents limitation/restrictions of space, where restrictions are only made perpendicular to the cartesian axes.

- **G4RotationMatrixStore** - a container for optionally storing created G4RotationMatrices.

- **G4SolidStore** - a container for optionally storing created solids. It enables traversal of all/any solids by the UI/user/etc. The class is a "singleton".

- **G4VSolid** - position independent geometrical entities. They have only 'shape', and encompass both CSG and boundary representations. They are optionally entered into the G4SolidStore. This class defines, but does not implement, functions to compute distances to/from the shape. Functions are also defined to check whether a point is inside the shape, to return the surface normal of the shape at a given point, and to compute the extent of the shape.

- **G4VSweptSolid** - a solid created by performing a 3D transformation on a finite planar face.

- **G4HalfSpaceSolid** - a solid created by the boolean AND of one or more half space surfaces.

- **G4BREPSolid** - a solid created by an abitrary set of finite surfaces.

- **G4VTouchable** - a class that maintains a "reference" on a given touchable element of the detector - a kind of bookmark. It enables a given detector element to be saved during tracking (in case of booleans/user code/etc.) and the corresponding G4PhysicalVolume retrieved later, with its "state" information (path through the tree) optionally restored so that navigation can be restarted. G4Touchables provide fast access to the transformation from the global reference system to that of the saved detector element.

- **G4TouchableHistory** - object representing a touchable detector element, and its history in the geomtrical hierarchy, including its net resultant local-¿global transform.

- **G4GRSSolid** - object representing a touchable solid. It maintains the association between a solid and its net resultant local-to-global transform.

- **G4GRSVolume** - object representing a touchable detector element. It maintains associations between a physical volume and its net resultant local-to-global transform.

- **G4TransformStore** - a container for optionally storing created G4AffineTransform objects. It is responsible for storing and providing access to transformations that are constant at tracking time.

- **G4AffineTransform** - a class for geometric affine transformations. It supports efficient arbitrary rotation and transformation of vectors and the computation of compound and inverse transformations. A "rotation flag" is maintained internally for greater computational efficiency for transforms that do not involve rotation.

- **G4UserLimits** - responsible for user limits on step size, ascribable to individual volumes.

Fig. 8.1 shows a general overview, in UML notation, of the geometry design. A detailed collection of class diagrams from the geometry category is found in the Appendix.

8.3 Status of this chapter

27.06.05 subsection on design philosphy (from Geant4 general paper) added by D.H. Wright

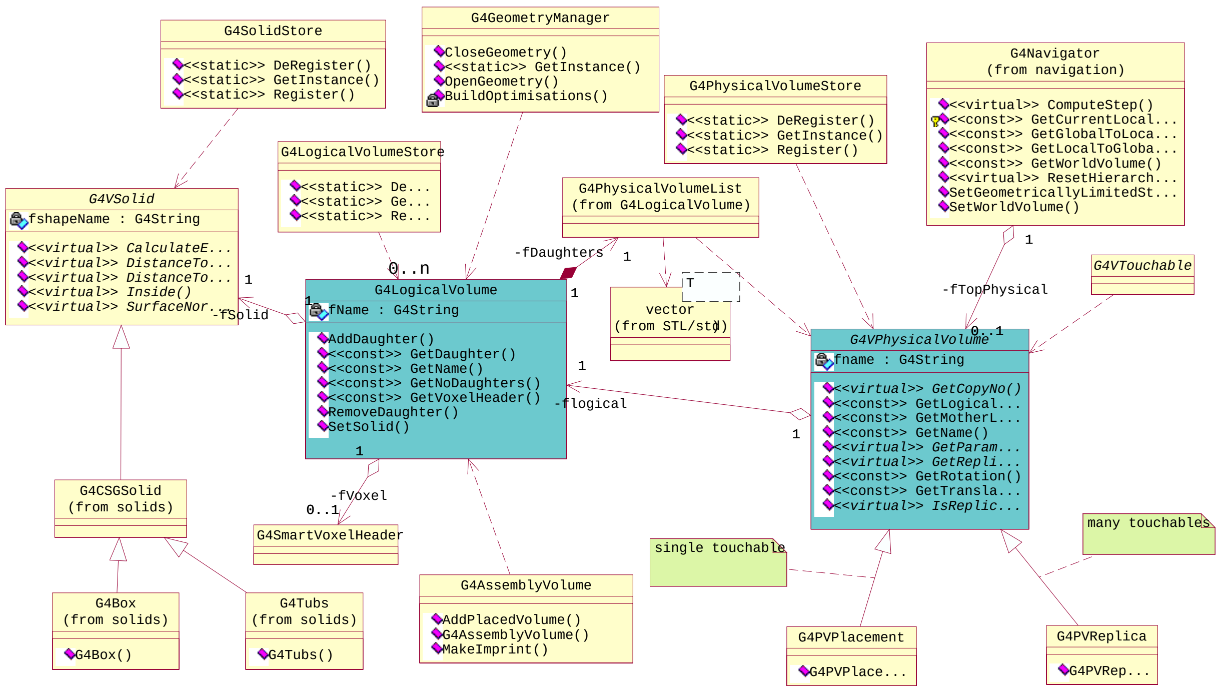


Figure 8.1: Overview of the geometry

Chapter 9

Electromagnetic Fields

The object-oriented design of the classes related to the electromagnetic field is shown in the class diagram of Fig. 9.1. The diagram is described in UML notation.

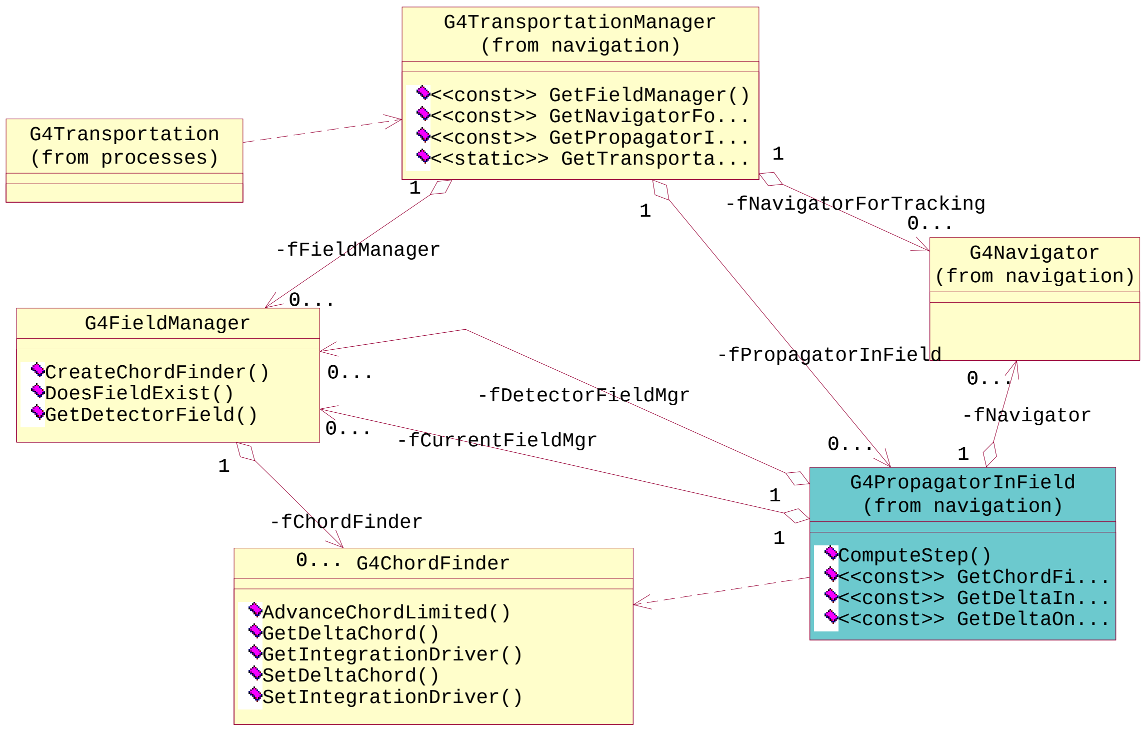


Figure 9.1: Electromagnetic Field

Chapter 10

Particles

10.1 Design Philosophy

The particles category implements the facilities necessary to describe the physical properties of particles for the simulation of particle-matter interactions. All particles are based on the *G4ParticleDefinition* class, which describes basic properties such as mass, charge, etc., and also allows the particle to carry a list of processes to which it is sensitive. A first-level extension of this class defines the interface for particles that carry cuts information, for example range cut versus energy cut equivalence. A set of virtual, intermediate classes for leptons, bosons, mesons, baryons, etc., allows the implementation of concrete particle classes which define the actual particle properties and, in particular, implement the actual range versus energy cuts equivalence. All concrete particle classes are instantiated as singletons to ensure that all physics processes refer to the same particle properties.

10.2 Class Design

The object-oriented design of the 'particles' related classes is shown in the following class diagrams. The diagrams are described in the Booch notation. Fig. 10.1 shows a general overview of the particle classes. Fig. 10.2 shows classes related to the particle table. Fig. 10.3 shows the classes related to the particle decay table.

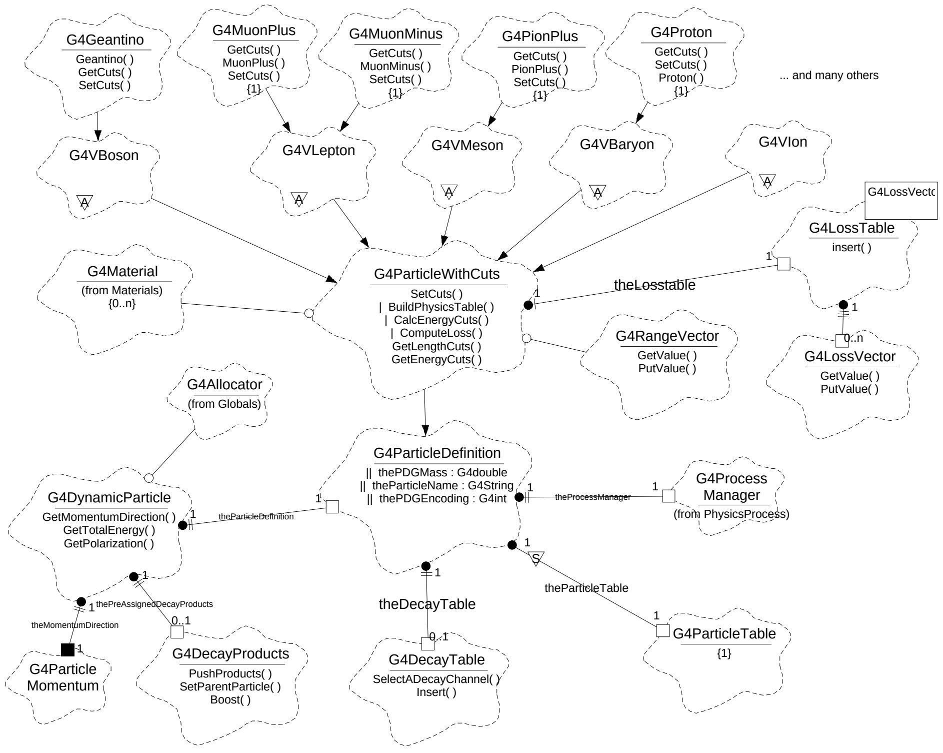


Figure 10.1: Particle classes

10.3 Status of this chapter

27.06.05 section on design philosophy added (from Geant4 general paper) by D.H. Wright

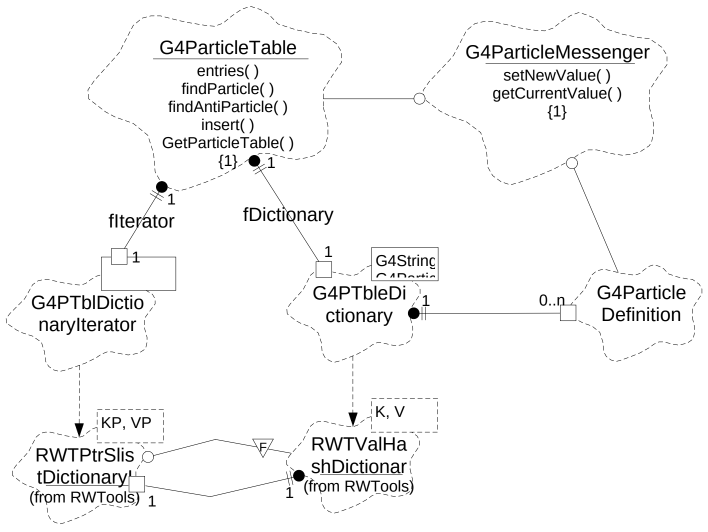


Figure 10.2: Particle Table

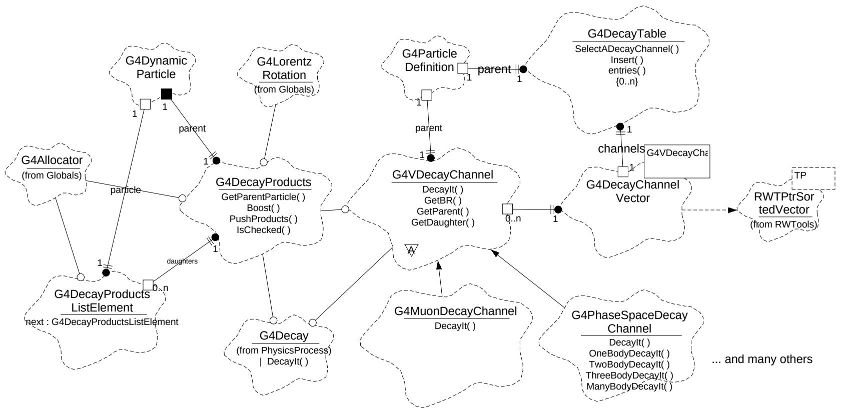


Figure 10.3: Particle Decay Table

Chapter 11

Materials

11.1 Design Philosophy

The design of the materials category reflects what exists in nature: materials are made of a single element or a mixture of elements, and elements are made of a single isotope or a mixture of isotopes. Because the physical properties of materials can be described in a generic way by quantities which can be specified directly, such as density, or derived from the element composition, only concrete classes are necessary in this category.

The material category implements the facilities necessary to describe the physical properties of materials for the simulation of particle-matter interactions. Characteristics like radiation and interaction length, excitation energy loss, coefficients in the Bethe-Bloch formula, shell correction factors, etc., are computed from the element, and if necessary, the isotope composition.

The material category also implements facilities to describe surface properties used in the tracking of optical photons.

11.2 Class Design

The object-oriented design of the 'materials' related classes is shown in the class diagram: Fig. 11.1. The diagram is described in the Booch notation.

11.3 Status of this chapter

27.06.05 section on design philosophy add (from Geant4 general paper) by D.H. Wright

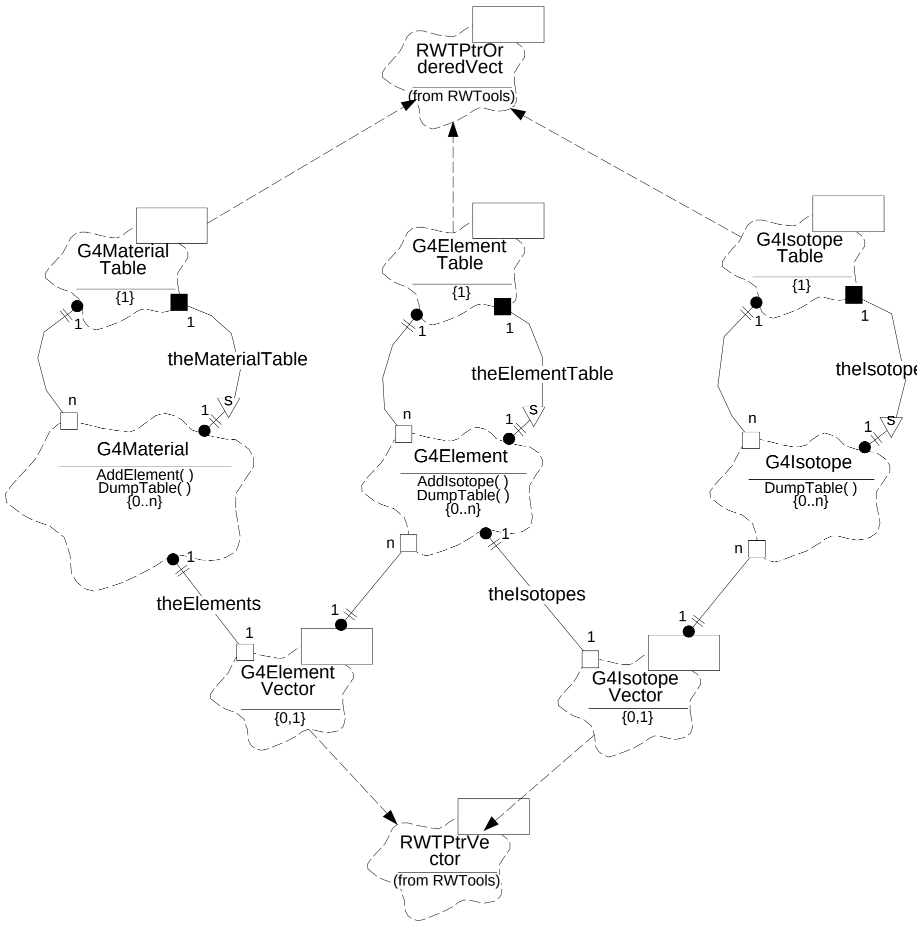


Figure 11.1: Materials

Chapter 12

Global Usage

12.1 Design Philosophy

The global category covers the system of units, constants, numerics and random number handling. It can be considered a place-holder for "general purpose" classes used by all categories defined in GEANT4. No back-dependencies to other GEANT4 categories affect the "global" domain. There are direct dependencies of the global category on external packages, such as CLHEP, STL, and miscellaneous system utilities.

Within the management sub-category are "utility" classes generally used within the GEANT4 kernel. They are, for the most part, uncorrelated with one another and include:

- *G4Allocator*
- *G4FastVector*
- *G4ReferenceCountedHandle*
- *G4PhysicsVector, G4LPhysicsFreeVector, G4PhysicsOrderedFreeVector*
- *G4Timer*
- *G4UserLimits*
- *G4UnitsTable*

A general description of these classes is given in section 3.2 of the GEANT4 User's Guide for Application Developers.

In applications where it is necessary to generate random numbers (normally from the same engine) in many different methods and parts of the program, it is highly desirable not to rely on or require knowledge of the global objects instantiated. By using static methods via a unique generator, the randomness of a sequence of numbers is best assured. Hence the use of a static generator has been introduced in the original design of HEPRandom as a project requirement in GEANT4.

12.2 Class Design

Analysis and design of the HEPRandom module have been achieved following the Booch Object-Oriented methodology. Some of the original design diagrams in Booch notation are reported below. Fig. 12.1 is a general picture of the static class diagram.

- **HepRandomEngine** - abstract class defining the interface for each Random engine. Its pure virtual methods must be defined by its subclasses representing the concrete Random engines.

- **HepJamesRandom** - class inheriting from HepRandomEngine and defining a flat random number generator according to the algorithm described in "F.James, Comp.Phys.Comm. 60 (1990) 329". This class is instantiated by default as the default random engine.

- **DRand48Engine** - class inheriting from HepRandomEngine and defining a flat random number generator according to the drand48() and srand48() system functions from the C standard library.

- **RandEngine** - class inheriting from HepRandomEngine and defining a flat random number generator according to the rand() and srand() system functions from the C standard library.

- **RanluxEngine** - class inheriting from HepRandomEngine and defining a flat random number generator according to the algorithm described in "F.James, Comp.Phys.Comm. 60 (1990) 329-344" and originally implemented in FORTRAN 77 as part of the MATHLIB HEP library. It provides 5 different "luxury" levels [0..4].

- **RanecuEngine** - class inheriting from HepRandomEngine and defining a flat random number generator according to the algorithm RANECU originally written in FORTRAN 77 as part of the MATHLIB HEP library. It uses a table of seeds which provides uncorrelated couples of seed values.

- **HepRandom** - the main class collecting all the methods defining the different random generators applied to HepRandomEngine. It is a singleton class which all the distribution classes derive from. This singleton is instantiated by default.
- **RandFlat** - distribution class for flat random number generation. It also provides methods to fill an array of flat random values, given its size or shoot bits.
- **RandExponential** - distribution class defining exponential random number distribution, given a mean. It also provides a method to fill an array of flat random values, given its size.
- **RandGauss** - distribution class defining Gauss random number distribution, given a mean or specifying also a deviation. It also provides a method to fill an array of flat random values, given its size.
- **RandBreitWigner** - distribution class defining the Breit-Wigner random number distribution. It also provides a method to fill an array of flat random values, given its size.
- **RandPoisson** - distribution class defining Poisson random number distribution, given a mean. It also provides a method to fill an array of flat random values, given its size.

Fig. 12.2 is a dynamic object diagram illustrating the situation when a single random number is thrown by the static generator according to one of the available distributions. Only one engine is assumed to active at a time.

Fig. 12.3 illustrates a random number being thrown by explicitly specifying an engine which can be shared by many distribution objects. The static interface is skipped here.

Fig. 12.4 illustrates the situation when many generators are defined, each by a distribution and an engine. The static interface is skipped here.

For detailed documentation about the HEPRandom classes see the CLHEP Reference Guide
(http://cern.ch/wwwasd/lhc++/clhep/manual/RefGuide/index.html)
or the CLHEP User Manual
(http://cern.ch/wwwasd/lhc++/clhep/manual/UserGuide/index.html).
Informations written in this manual are extracted from the original manifesto distributed with the HEPRandom package
(http://cern.ch/wwwasd/geant/geant4_public/Random.html).

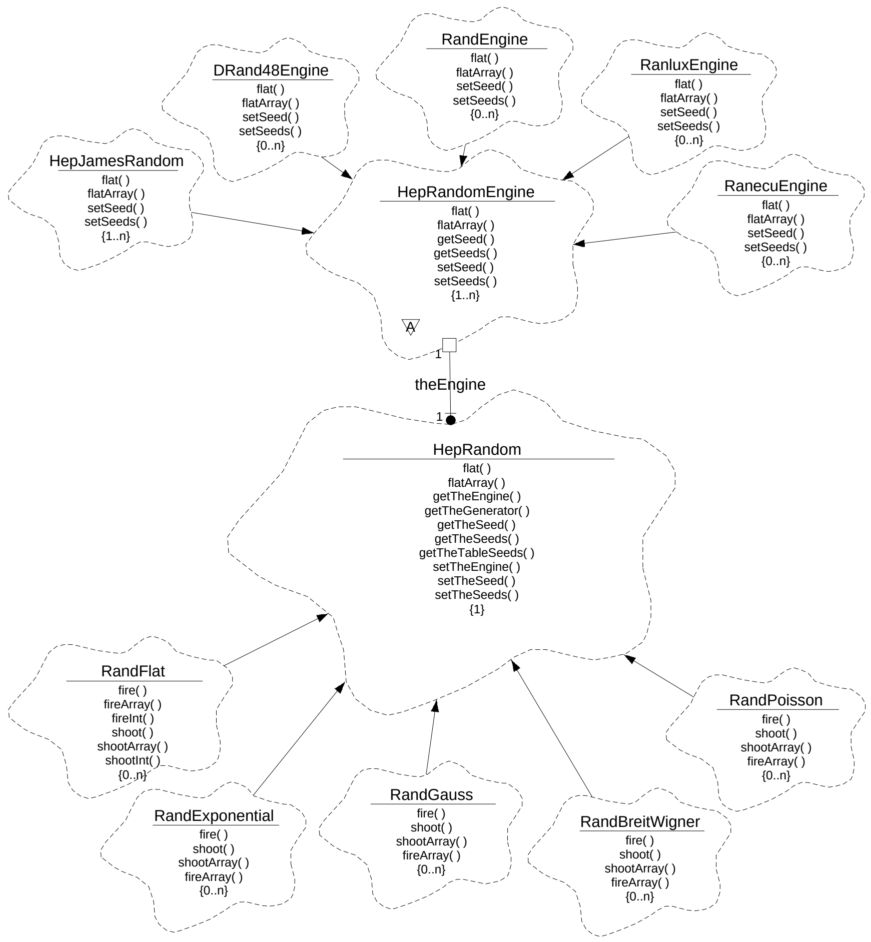


Figure 12.1: HEPRandom module

HEPNumerics The HEPNumerics module includes a set of classes which implement numerical algorithms for general use in GEANT4. Section 3.2.3 of the User's Guide for Application Developers contains a description of each class. Most of the algorithms were implemented using methods from the following books:

- B.H. Flowers, "An introduction to Numerical Methods In C++", Claredon Press, Oxford 1995.
- M. Abramowitz, I. Stegun, "Handbook of mathematical functions", DOVER Publications INC, New York 1965 ; chapters 9, 10, and 22.

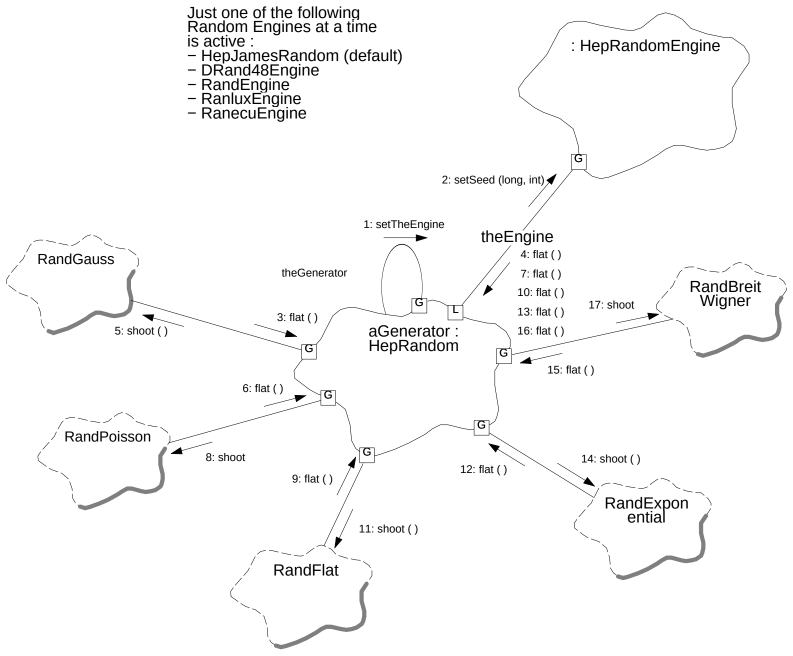


Figure 12.2: Shooting via the generator

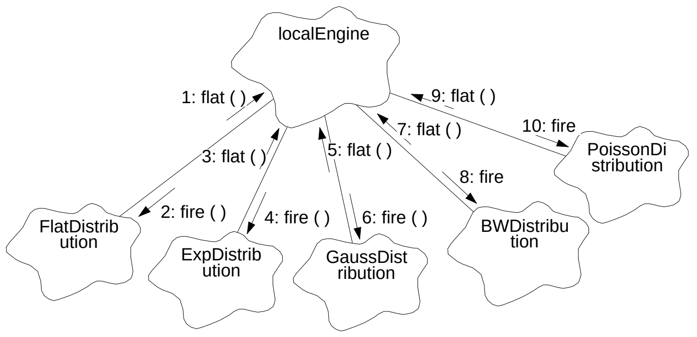


Figure 12.3: Shooting via distribution objects

HEPGeometry Documentation for the HEPGeometry module is provided in the CLHEP Reference Guide (http://cern.ch/wwwasd/lhc++/clhep/manual/RefGuide/index.html) or the CLHEP User Manual (http://cern.ch/wwwasd/lhc++/clhep/manual/UserGuide/index.html)

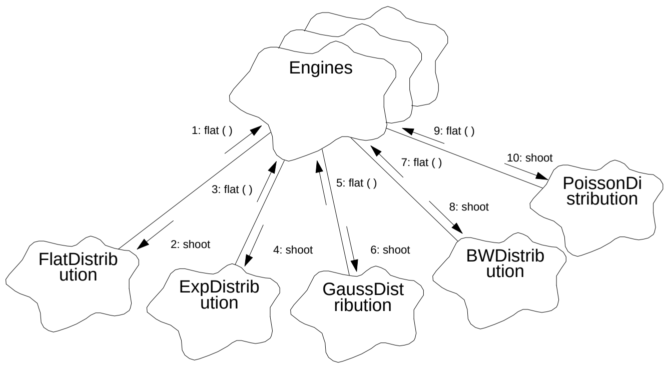


Figure 12.4: Shooting with arbitrary engines

12.3 Status of this chapter

01.12.02 minor update by G. Cosmo
18.06.05 introductory paragraphs added and minor grammar changes by D.H. Wright

Chapter 13

Visualization

13.1 Design Philosophy

The visualization category consists of the classes required to display detector geometry, particle trajectories, tracking steps, and hits. It also provides visualization drivers, which are interfaces to external graphics systems.

A wide variety of user requirements went into the design of the visualization category, for example:

- very quick response in surveying successive events,
- high-quality output for presentation and documentation,
- flexible camera control for debugging detector geometry and physics,
- selection of visualizable objects,
- interactive picking of graphical objects for attribute editing or feedback to the associated data,
- highlighting incorrect intersections of physical volumes,
- co-working with graphical user interfaces.

Because it is very difficult to respond to all of these requirements with only one built-in visualizer, an abstract interface was developed which supports several complementary graphics systems. Here the term "graphics system" means either an application running as a process independent of GEANT4 or a graphics library to be compiled with GEANT4. A concrete implementation of the interface is called a visualization driver, which can use a graphics library directly, communicate with an independent process via pipe or socket, or simply write an intermediate file for a separate viewer.

13.2 Class Design

- **G4VVisManager** - abstract base class for visualization managers.
- **G4VisManager** - controls GEANT4 visualization. This class creates and registers graphics systems, scenes, scene handlers and viewers and manages them. It is a singleton and an abstract class, requiring the user to derive from it a concrete visualization manager. Roles and structure of the visualization manager are described in Chapter 8 of the User's Guide for Application Developers.
- **G4VisExecutive** - concrete visualization manager that implements the virtual function RegisterGraphicsSystems.
- **G4VGraphicsSystem, G4VSceneHandler, G4VView** - abstract interface classes for initialization of a graphics system, modeling 3D scenes, and rendering the modeled 3D scenes, respectively. By defining a set of three C++ classes inheriting from these virtual base classes, an arbitrary graphics system can easily be plugged in to GEANT4. The plugged-in graphics system is then available for visualizing detector simulations.

The set of three concrete classes mentioned in the last item is called a "visualization driver". The DAWN-File driver, for example, is the interface to the Fukui Renderer DAWN, and is implemented by the following set of classes:

1. G4DAWNFILE : public G4VGraphicsSystem
   for initialization
2. G4DAWNFILESceneHandler : public G4VSceneHandler
   for modeling 3D scenes
3. G4DAWNFILEView : public G4VView
   for rendering 3D scenes

Several visualization drivers are already prepared by default. They are complementary to each other in many aspects. For details, see Chapter 8 of the User's Guide for Application Developers.

To create a new graphics system for GEANT4, it is necessary to implement a new set of three classes composing a new visualization driver. A skeleton set of classes is included in the visualization category under subdirectory visualization/XXX (but not normally registered). A recommended approach is to copy the contents of XXX to a new subdirectory with a name that suits your new system. Then

1. Change the name of the files (change XXX to something that suits your system).

2. Change XXX similarly in all files.

3. Change XXX similarly in "name := G4XXX" in GNUmakefile.

4. Add your new subdirectory to SUBDIRS and SUBLIBS in visualization/GNUmakefile.

5. Look at the code and use it to build your graphics system. You might also find it useful to look at ASCIITree (and VTree) as an example of a minimal graphics system. Look at FukuiRenderer as an example of a system which implements AddSolid methods for some solids. Look at OpenGL as an example of a system which implements a graphical database (display lists) and the machinery to decide when to rebuild. (OpenGL is complicated by the proliferation of combinations of the use or not of display lists for three window systems, X-windows, X with motif (interactive), Microsoft Windows (Win32), a total of six combinations, and much use is made of inheritance to avoid code duplication.)

6. If it requires external libraries, introduce two new environment variables G4VIS_BUILD_XXX_DRIVER and G4VIS_USE_XXX (where XXX is your choice as above) and make the modifications to:

   - source/visualization/management/include/G4VisExecutive.icc
   - source/visualization/management/src/G4VisManager.cc
   - config/G4VIS_BUILD.gmk
   - config/G4VIS_USE.gmk

13.3 Modeling sub-category

- **G4VModel** - a base class for visualization models. A model is a graphics-system-indepedent description of a Geant4 component.

The sub-category visualization/modeling defines how to model a 3D scene for visualization. The term "3D scene" indicates a set of visualizable component objects put in a 3D world. A concrete class inheriting from the abstract base class G4VModel defines a "model", which describes how to visualize the corresponding component object belonging to a 3D scene. G4ModelingParameters defines various associated parameters.

For example, G4PhysicalVolumeModel knows how to visualize a physical volume. It describes a physical volume and its daughters to any desired depth. G4HitsModel knows how to visualize hits. G4TrajectoriesModel knows how to visualize trajectories. G4FlavoredParallelWorld knows how to visualize flavoured parallel world volumes.

The main task of a model is to describe itself to a 3D scene by giving a concrete implementation of the following virtual method of G4VModel:

```
virtual void DescribeYourselfTo (G4VGraphicsScene&) = 0;
```

The argument class G4VGraphicsScene is a minimal abstract interface of a 3D scene for the GEANT4 kernel defined in the graphics_reps category. Since G4VSceneHandler and its concrete descendants inherit from G4VGraphicsScene, the method DescribeYourselfTo() can pass information of a 3D scene to a visualization driver.

It is easy for a toolkit developer of GEANT4 to add a new kind of visualizable component object. It is done by implementing a new class inheriting from G4VModel.

13.4 View parameters

View parameters such as camera parameters, drawing styles (wireframe/surface etc) are held by G4ViewParameters. Each viewer holds a view parameters object which can be changed interactively and a default object (for use in the `/vis/viewer/reset` command).

If a toolkit developer of GEANT4 wants to add entries of view parameters, he should add fields and methods to G4ViewParameters.

13.5 Status of this section

27.06.05 partially re-organized and section on design philosophy added (from Geant4 general paper) by D.H. Wright

Chapter 14

Intercoms

14.1 Design Philosophy

The intercoms category implements an expandable command interpreter which is the key mechanism in GEANT4 for realizing customizable and state-dependent user interactions with all categories without being perturbed by the dependencies among classes. The capturing of commands is handled by a C++ abstract class *G4UIsession*. Various concrete implementations of the command capturer are contained in the [user] interfaces category. Taking into account the rapid evolution of graphical user interface (GUI) technology and consequent dependence on external facilities, plural and extensible GUIs are offered.

Programmers need only know how to register the commands and parameters appropriate to their problem domain; no knowledge of GUI programming is required to allow an application to use them through one of the available GUIs.

The intercoms category also provides the virtual base classes

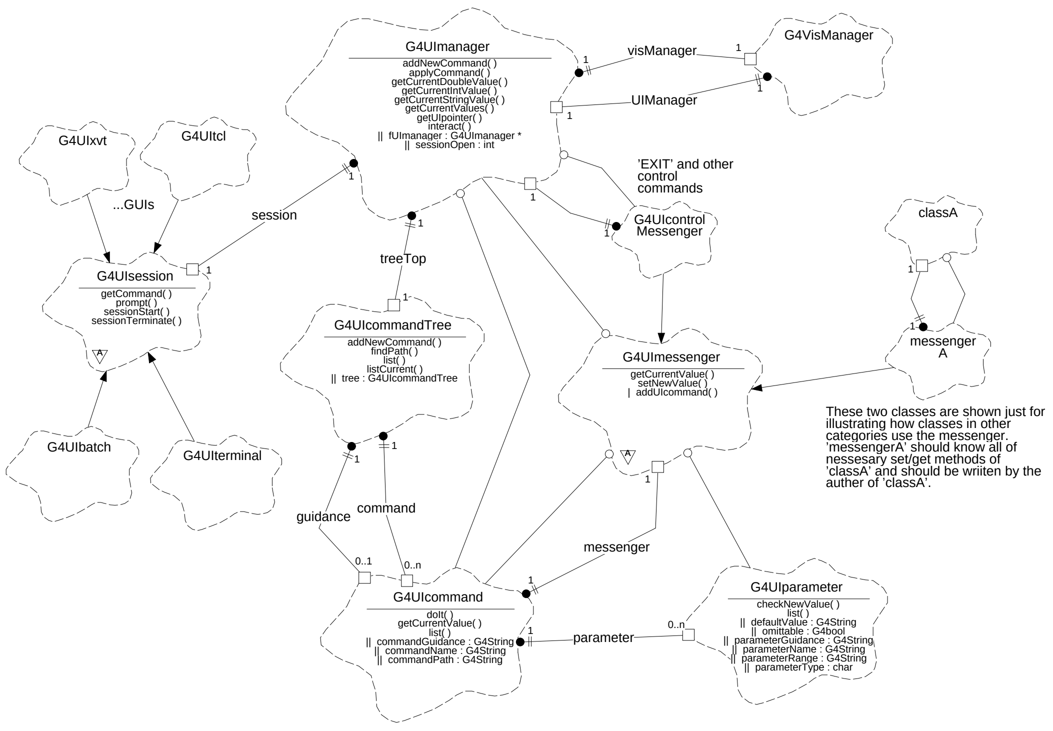
- G4VVisManager,
- G4VGraphicsScene, and
- G4VGlobalFastSimulationManager.

14.2 Class Design

- **G4UISession** -
- **G4UIBatch** -

- **G4UICommand** -
- **G4UIparameter** -
- **G4UImessenger** -

The object-oriented design of the 'user interface' related classes is shown in the class diagram Fig. 14.1. The diagram is described in the Booch notation.



These two classes are shown just for illustrating how classes in other categories use the messenger. 'messengerA' should know all of nessesary set/get methods of 'classA' and should be wriiten by the auther of 'classA'.

Figure 14.1: Overview of intercom classes

14.3 Status of this section

27.06.05 design philosophy (from Geant4 general paper) and class design sections added by D.H. Wright

Part III

Extending Toolkit Functionality

Chapter 15

Geometry

15.1 What can be extended ?

GEANT4 already allows a user to describe any desired solid, and to use it in a detector description, in some cases, however, the user may want or need to extend GEANT4's geometry. One reason can be that some methods and types in the geometry are general and the user can utilise specialised knowledge about his or her geometry to gain a speedup. The most evident case where this can happen is when a particular type of solid is a key element for a specific detector geometry and an investment in improving its runtime performance may be worthwhile.

To extend the functionality of the Geometry in this way, a toolkit developer must write a small number of methods for the new solid. We will document below these methods and their specifications. Note that the implementation details for some methods are not a trivial matter: these methods must provide the functionality of finding whether a point is inside a solid, finding the intersection of a line with it, and finding the distance to the solid along any direction. However once the solid class has been created with all its specifications fulfilled, it can be used like any GEANT4 solid, as it implements the abstract interface of G4VSolid.

Other additions can also potentially be achieved. For example, an advanced user could add a new way of creating physical volumes. However, because each type of volume has a corresponding navigator helper, this would require to create a new Navigator as well. To do this the user would have to inherit from G4Navigator and modify the new Navigator to handle this type of volumes. This can certainly be done, but will probably be made easier to achieve in the future versions of the GEANT4 toolkit.

15.2 Adding a new type of Solid

We list below the required methods for integrating a new type of solid in GEANT4. Note that GEANT4's specifications for a solid pay significant attention to what happens at points that are within a small distance (tolerance, *kCarTolerance* in the code) of the surface. So special care must be taken to handle these cases in considering all different possible scenarios, in order to respect the specifications and allow the solid to be used correctly by the other components of the geometry module.

Creating a derived class of G4VSolid The solid must inherit from G4VSolid or one of its derived classes and implement its virtual functions.

Mandatory member functions you must define are the following pure virtual of G4VSolid:

```
EInside Inside(const G4ThreeVector& p)
G4double DistanceToIn(const G4ThreeVector& p)
G4double DistanceToIn(const G4ThreeVector& p, const G4ThreeVector& v)
G4ThreeVector SurfaceNormal(const G4ThreeVector& p)
G4double DistanceToOut(const G4ThreeVector& p)
G4double DistanceToOut(const G4ThreeVector& p, const G4ThreeVector& v,
                       const G4bool calcNorm=false,
                             G4bool *validNorm=0, G4ThreeVector *n)
G4bool CalculateExtent(const EAxis pAxis,
                       const G4VoxelLimits& pVoxelLimit,
                       const G4AffineTransform& pTransform,
                             G4double& pMin,
                             G4double& pMax) const
G4GeometryType GetEntityType() const
std::ostream& StreamInfo(std::ostream& os) const
```

They must perform the following functions

```
EInside Inside(const G4ThreeVector& p)
```

This method must return:

- kOutside if the point at offset p is outside the shape boundaries plus Tolerance/2,
- kSurface if the point is <= Tolerance/2 from a surface, or
- kInside otherwise.

```
G4ThreeVector SurfaceNormal(const G4ThreeVector& p)
```

Return the outwards pointing unit normal of the shape for the surface closest to the point at offset p.

```
G4double DistanceToIn(const G4ThreeVector& p)
```

Calculate distance to nearest surface of shape from an outside point p. The distance can be an underestimate.

```
G4double DistanceToIn(const G4ThreeVector& p, const G4ThreeVector& v)
```

Return the distance along the normalised vector v to the shape, from the point at offset p. If there is no intersection, return kInfinity. The first intersection resulting from 'leaving' a surface/volume is discarded. Hence, this is tolerant of points on surface of shape.

```
G4double DistanceToOut(const G4ThreeVector& p)
```

Calculate distance to nearest surface of shape from an inside point. The distance can be an underestimate.

```
G4double DistanceToOut(const G4ThreeVector& p, const G4ThreeVector& v,
                       const G4bool calcNorm=false,
                       G4bool *validNorm=0, G4ThreeVector *n=0);
```

Return distance along the normalised vector v to the shape, from a point at an offset p inside or on the surface of the shape. Intersections with surfaces, when the point is not greater than kCarTolerance/2 from a surface, must be ignored.

If calcNorm is true, then it must also set validNorm to either

- true, if the solid lies entirely behind or on the exiting surface. Then it must set n to the outwards normal vector (the Magnitude of the vector is not defined).
- false, if the solid does not lie entirely behind or on the exiting surface.

If calcNorm is false, then validNorm and n are unused.

```
G4bool CalculateExtent(const EAxis pAxis,
                       const G4VoxelLimits& pVoxelLimit,
                       const G4AffineTransform& pTransform,
                             G4double& pMin,
                             G4double& pMax) const
```

Calculate the minimum and maximum extent of the solid, when under the specified transform, and within the specified limits. If the solid is not intersected by the region, return false, else return true.

```
G4GeometryType GetEntityType() const;
```

Provide identification of the class of an object (required for persistency and STEP interface).

```
std::ostream& StreamInfo(std::ostream& os) const
```

Should dump the contents of the solid to an output stream.
The method:

```
G4double GetCubicVolume()
```

should be implemented for every solid in order to cache the computed value (and therefore reuse it for future calls to the method) and to eventually implement a precise computation of the solid's volume. If the method will not be overloaded, the default implementation from the base class will be used (estimation through a Monte Carlo algorithm) and the computed value will not be stored.

There are a few member functions to be defined for the purpose of visualisation. The first method is mandatory, and the next four are not.

```
  // Mandatory
  virtual void DescribeYourselfTo (G4VGraphicsScene& scene) const = 0;

  // Not mandatory
  virtual G4VisExtent GetExtent() const;
  virtual G4Polyhedron* CreatePolyhedron () const;
  virtual G4NURBS*      CreateNURBS      () const;
  virtual G4Polyhedron* GetPolyhedron    () const;
```

What these methods should do and how they should be implemented is described here.

```
 void DescribeYourselfTo (G4VGraphicsScene& scene) const;
```

This method is required in order to identify the solid to the graphics scene. It is used for the purposes of "double dispatch". All implementations should be similar to the one for G4Box:

```
void G4Box::DescribeYourselfTo (G4VGraphicsScene& scene) const
{
  scene.AddSolid (*this);
}
```

The method:

```
 G4VisExtent GetExtent() const;
```

provides extent (bounding box) as a possible hint to the graphics view. You must create it by finding a box that encloses your solid, and returning a VisExtent that is created from this. The G4VisExtent must presumably be given the minus x, plus x, minus y, plus y, minus z and plus z extents of this "box". For example a cylinder can say

```
G4VisExtent G4Tubs::GetExtent() const
{
  // Define the sides of the box into which the G4Tubs instance would fit.
  return G4VisExtent (-fRMax, fRMax, -fRMax, fRMax, -fDz, fDz);
}
```

The method:

```
 G4Polyhedron* CreatePolyhedron () const;
```

is required by the visualisation system, in order to create a realistic rendering of your solid. To create a G4Polyhedron for your solid, consult G4Polyhedron.

While the method:

```
 G4Polyhedron* GetPolyhedron () const;
```

is a "smart" access function that creates on requests a polyhedron and stores it for future access and should be customised for every solid.

The method:

```
 G4NURBS* CreateNURBS () const;
```

is not currently utilised, so you do not have to implement it.

15.3 Modifying the Navigator

For the vast majority of use-cases, it is not indeed necessary (and definitely not advised) to extend or modify the existing classes for navigation in the geometry. A possible use-case for which this may apply, is for the description of a new kind of physical volume to be integrated. We believe that our set of choices for creating physical volumes is varied enough for nearly all needs. Future extensions of the GEANT4 toolkit will probably make easier exchanging or extending the G4Navigator, by introducing an abstraction level simplifying the customisation. At this time, a simple abstraction level of the navigator is provided by allowing overloading of the relevant functionalities.

Extending the Navigator The main responsibilities of the Navigator are:

- locate a point in the tree of the geometrical volumes;
- compute the length a particle can travel from a point in a certain direction before encountering a volume boundary.

The Navigator utilises one helper class for each type of physical volume that exists. You will have to reuse the helper classes provided in the base Navigator or create new ones for the new type of physical volume.

To extend G4Navigator you will have then to inherit from it and modify these functions in your ModifiedNavigator to request the answers for your new physical volume type from the new helper class. The ModifiedNavigator should delegate other cases to the GEANT4's standard Navigator.

Replacing the Navigator Replacing the Navigator is another possible operation. It is similar to extending the Navigator, in that any types of physical volume that will be allowed must be handled by it. The same functionality is required as described in the previous section.

However the amount of work is probably potentially larger, if support for all the current types of physical volumes is required.

The Navigator utilises one helper class for each type of physical volume that exists. These could also potentially be replaced, allowing a simpler way to create a new navigation system.

Chapter 16

Electromagnetic Fields

16.1 Creating a New Type of Field

GEANT4 currently handles magnetic and electric fields and, in future releases, will handle combined electromagnetic fields. Fields due to other forces, not yet included in GEANT4, can be provided by describing the new field and the force it exerts on a particle passing through it. For the time being, all fields must be time-independent. This restriction may be lifted in the future.

In order to accommodate a new type of field, two classes must be created: a field type and a class that determines the force. The GEANT4 system must then be informed of the new field.

A new Field class A new type of Field class may be created by inheriting from G4Field

```
class NewField : public G4Field
{
   public:
      void  GetFieldValue( const  double Point[3],
                                  double *pField )=0;
}
```

and deciding how many components your field will have, and what each component represents. For example, three components are required to describe a vector field while only one component is required to describe a scalar field.

If you want your field to be a combination of different fields, you must choose your convention for which field goes first, which second etc. For example, to define an electromagnetic field we follow the convention that components 0,1 and 2 refer to the magnetic field and components 3, 4 and 5 refer to the electric field.

By leaving the GetFieldValue method pure virtual, you force those users who want to describe their field to create a class that implements it for their detector's instance of this field. So documenting what each component means is required, to give them the necessary information.

For example someone can describe DetectorAbc's field by creating a class DetectorAbcField, that derives from your NewField

```
class DetectorAbcField : public NewField
{
  public:
     void  MyFieldGradient::GetFieldValue( const double Point[3],
                                           double *pField );
}
```

They then implement the function GetFieldValue

```
      void  MyFieldGradient::GetFieldValue( const  double Point[3],
                                            double *pField )
      {
         // We expect pField to point to pField[9];
         // This & the order of the components of pField is your own
         // convention

         // We calculate the value of pField at Point ...
      }
```

A new Equation of Motion for the new Field Once you have created a new type of field, you must create an Equation of Motion for this Field. This is required in order to obtain the force that a particle feels.

To do this you must inherit from G4Mag_EqRhs and create your own equation of motion that understands your field. In it you must implement the virtual function EvaluateRhsGivenB. Given the value of the field, this function calculates the value of the generalised force. This is the only function that a subclass must define.

```
     virtual void EvaluateRhsGivenB( const  G4double y[],
                                     const  G4double B[3],
                                            G4double dydx[] ) const = 0;
```

In particular, the derivative vector dydx is a vector with six components. The first three are the derivative of the position with respect to the curve length. Thus they should set equal to the normalised velocity, which is components 3, 4 and 5 of y.

```
(dydx[0], dydx[1], dydx[2]) = (y[3], y[4], y[5])
```

The next three components are the derivatives of the velocity vector with respect to the path length. So you should write the "force" components for

```
    dydx[3], dydx[4] and dydx[5]
```

for your field.

Get a G4FieldManager to use your field In order to inform the GEANT4 system that you want it to use your field as the global field, you must do the following steps:

1. Create a Stepper of your choice:

```
      yourStepper = new G4ClassicalRK( yourEquationOfMotion );
                  // or if your field is not smooth eg
                  //     new G4ImplicitEuler( yourEquationOfMotion );
```

2. Create a chord finder that uses your Field and Stepper. You must also give it a minimum step size, below which it does not make sense to attempt to integrate:

```
      yourChordFinder= new G4ChordFinder( yourField,
                                   yourMininumStep, // say 0.01*mm
                                   yourStepper );
```

3. Next create a G4FieldManager and give it that chord finder,

```
      yourFieldManager= new G4FieldManager();
      yourFieldManager.SetChordFinder(yourChordFinder);
```

4. Finally we tell the Geometry that this FieldManager is responsible for creating a field for the detector.

```
      G4TransportationManager::GetTransportationManager()
                             -> SetFieldManager( yourFieldManager );
```

Changes for non-electromagnetic fields If the field you are interested in simulating is not electromagnetic, another minor modification may be required. The transportation currently chooses whether to propagate a particle in a field or rectilinearly based on whether the particle is charged or not. If your field affects non-charged particles, you must inherit from the G4Transportation and re-implement the part of GetAlongStepPhysicalInteractionLength that decides whether the particles is affected by your force.

In particular the relevant section of code does the following:

```
  //  Does the particle have an (EM) field force exerting upon it?
//
if( (particleCharge!=0.0) ){

   fieldExertsForce= this->DoesGlobalFieldExist();
   // Future: will/can also check whether current volume's field is Zero or
   //  set by the user (in the logical volume) to be zero.
}
```

and you want it to ask whether it feels your force. If, for the sake of an example, you wanted to see the effects of gravity on a heavy hypothetical particle, you could say

```
//  Does the particle have my field's force exerted on it?
//
if (particle->GetName() == "VeryHeavyWIMP") {
   fieldExertsForce= this->DoesGlobalFieldExist();  // For gravity
}
```

After doing all these steps, you will be able to see the effects of your force on a particle's motion.

16.2 Status of this chapter

10.06.02 partially re-written by D.H. Wright
14.11.02 spell check by P. Arce

Chapter 17

Physics Processes

Adding a new electromagnetic process. Adding a new hadronic process.

17.1 Status of this chapter

27.06.05 under construction

Chapter 18

Hadronic Physics

18.1 Introduction

Optimal exploitation of hadronic final states played a key role in successes of all recent collider experiment in HEP, and the ability to use hadronic final states will continue to be one of the decisive issues during the analysis phase of the LHC experiments. Monte Carlo programs like GEANT4[1] facilitate the use of hadronic final states, and have been developed for many years.

We give an overview of the Object Oriented frameworks for hadronic generators in GEANT4, and illustrate the physics models underlying hadronic shower simulation on examples, including the three basic types of modeling; data-driven, parametrisation-driven, and theory-driven modeling, and their possible realisations in the Object Oriented component system of GEANT4. We put particular focus on the level of extendibility that can and has been achieved by our Russian dolls approach to Object Oriented design, and the role and importance of the frameworks in a component system.

18.2 Principal Considerations

The purpose of this section is to explain the implementation frameworks used in and provided by GEANT4 for hadronic shower simulation as in the 1.1 release of the program. The implementation frameworks follow the Russian dolls approach to implementation framework design. A top-level, very abstracting implementation framework provides the basic interface to the other GEANT4 categories, and fulfils the most general use-case for hadronic shower simulation. It is refined for more specific use-cases by implementing a hierarchy of implementation frameworks, each level implementing the common logic of a particular use-case package in a concrete implementation of the interface specification of one framework level above, this way refining the granularity of abstraction and delegation. This defines the Russian dolls architectural pattern. Abstract classes are used as the delegation mechanism[1]. All framework functional requirements were obtained through use-case analysis. In the following we present for each framework level the compressed use-cases, requirements, designs including the flexibility provided, and illustrate the framework functionality with examples. All design patterns cited are to be read as defined in [2].

18.3 Level 1 Framework - processes

There are two principal use-cases of the level 1 framework. A user will want to choose the processes used for his particular simulation run, and a physicist will want to write code for processes of his own and use these together with the rest of the system in a seamless manner.

Requirements

1. Provide a standard interface to be used by process implementations.
2. Provide registration mechanisms for processes.

Design and interfaces Both requirements are implemented in a sub-set of the tracking-physics interface in GEANT4. The class diagram is shown in figure 18.1.

All processes have a common base-class `G4VProcess`, from which a set of specialised classes are derived. Three of them are used as base classes for hadronic processes for particles at rest (`G4VRestProcess`), for interactions in flight (`G4VDiscreteProcess`), and for processes like radioactive decay where the same implementation can represent both these extreme cases (`G4VRestDiscreteProcess`).

Each of these classes declares two types of methods; one for calculating the time to interaction or the physical interaction length, allowing tracking to request the information necessary to decide on the process responsible for final state production, and one to compute the final state. These are pure virtual methods, and have to be implemented in each individual derived class, as enforced by the compiler.

[1]The same can be achieved with template specialisations with slightly improved CPU performance but at the cost of significantly more complex designs and, with present compilers, significantly reduced portability.

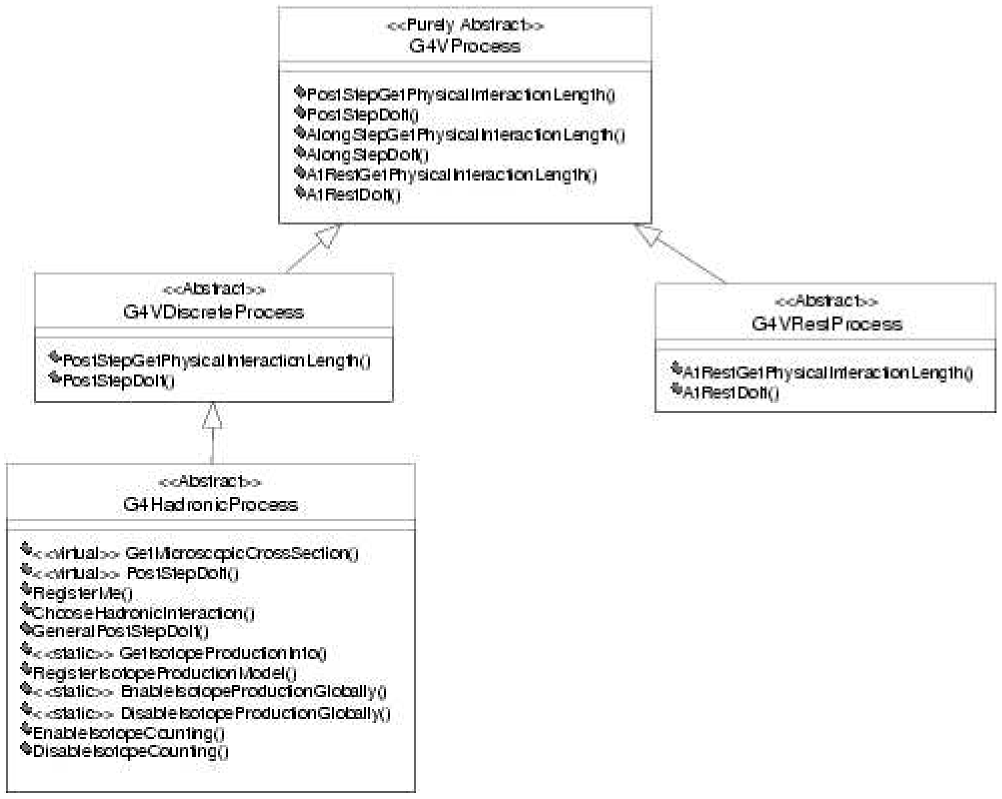

<<Purely Abstract>>
G4VProcess
PostStepGetPhysicalInteractionLength()
PostStepDoIt()
AlongStepGetPhysicalInteractionLength()
AlongStepDoIt()
AtRestGetPhysicalInteractionLength()
AtRestDoIt()
<<Abstract>>
G4VDiscreteProcess
PostStepGetPhysicalInteractionLength()
PostStepDoIt()
<<Abstract>>
G4VRestProcess
AtRestGetPhysicalInteractionLength()
AtRestDoIt()
<<Abstract>>
G4HadronicProcess
<<virtual>> GetMicroscopicCrossSection()
<<virtual>> PostStepDoIt()
RegisterMe()
ChooseHadronicInteraction()
GeneralPostStepDoIt()
<<static>> GetIsotopeProductionInfo()
RegisterIsotopeProductionModel()
<<static>> EnableIsotopeProductionGlobally()
<<static>> DisableIsotopeProductionGlobally()
EnableIsotopeCounting()
DisableIsotopeCounting()

Figure 18.1: Level 1 implementation framework of the hadronic category of Geant4.

Framework functionality The functionality provided is through the use of process base-class pointers in the tracking-physics interface, and the `G4ProcessManager`. All functionality is implemented in abstract, and registration of derived process classes with the `G4ProcessManager` of an individual particle allows for arbitrary combination of both GEANT4 provided processes, and user-implemented processes. This registration mechanism is a modification on a Chain of Responsibility. It is outside the scope of the current paper, and its description is available from [3].

18.4 Level 2 Framework - Cross Sections and Models

At the next level of abstraction, only processes that occur for particles in flight are considered. For these, it is easily observed that the sources of cross sections and final state production are rarely the same. Also, different sources will come with different restrictions. The principal use-cases of the framework are addressing these commonalities. A user might want to combine different cross sections and final state or isotope production models as provided by GEANT4, and a physicist might want to implement his own model for particular situation, and add cross-section data sets that are relevant for his particular analysis to the system in a seamless manner.

Requirements

1. Flexible choice of inclusive scattering cross-sections.
2. Ability to use different data-sets for different parts of the simulation, depending on the conditions at the point of interaction.
3. Ability to add user-defined data-sets in a seamless manner.
4. Flexible, unconstrained choice of final state production models.
5. Ability to use different final state production codes for different parts of the simulation, depending on the conditions at the point of interaction.
6. Ability to add user-defined final state production models in a seamless manner.
7. Flexible choice of isotope production models, to run in parasitic mode to any kind of transport models.

8. Ability to use different isotope production codes for different parts of the simulation, depending on the conditions at the point of interaction.

9. Ability to add user-defined isotope production models in a seamless manner.

Design and interfaces The above requirements are implemented in three framework components, one for cross-sections, final state production, and isotope production each. The class diagrams are shown in figure 18.2 for the cross-section aspects, figure 18.3 for the final state production aspects, and figure 18.4 for the isotope production aspects.

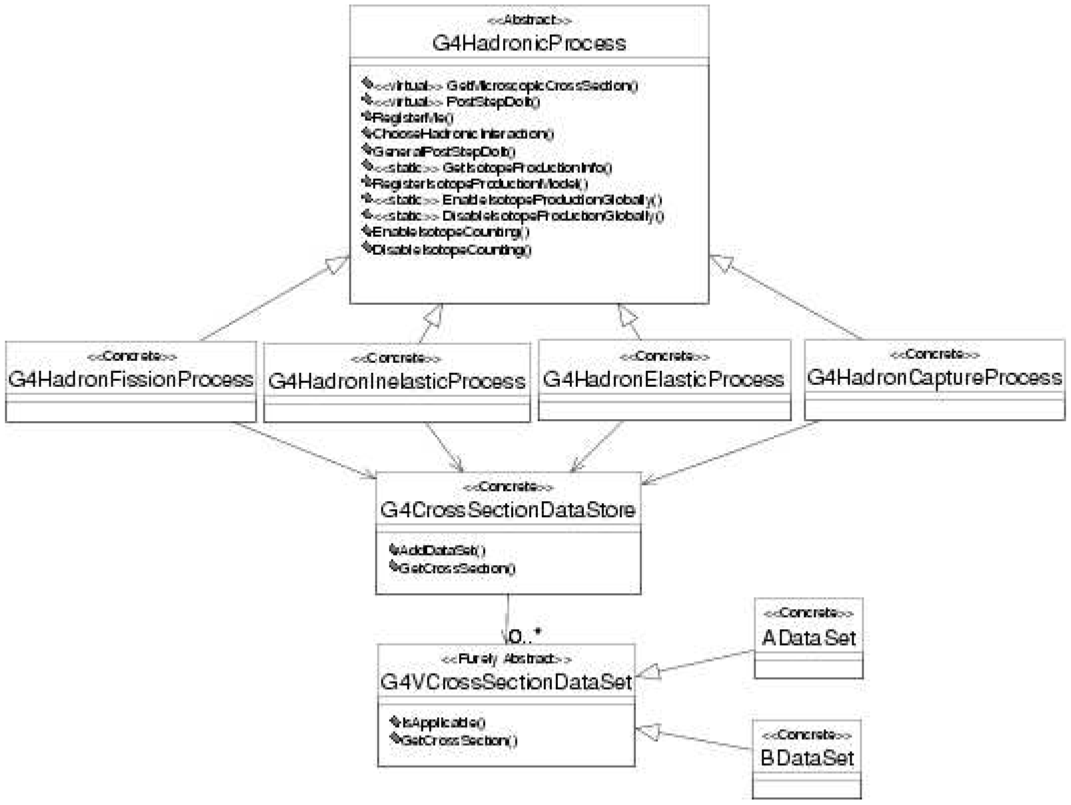


Figure 18.2: Level 2 implementation framework of the hadronic category of GEANT4; cross-section aspect.

The three parts are integrated in the `G4HadronicProcess` class, that serves as base-class for all hadronic processes of particles in flight.

Cross-sections Each hadronic process is derived from `G4HadronicProcess`, which holds a list of "cross section data sets". The term "data set" is repre-

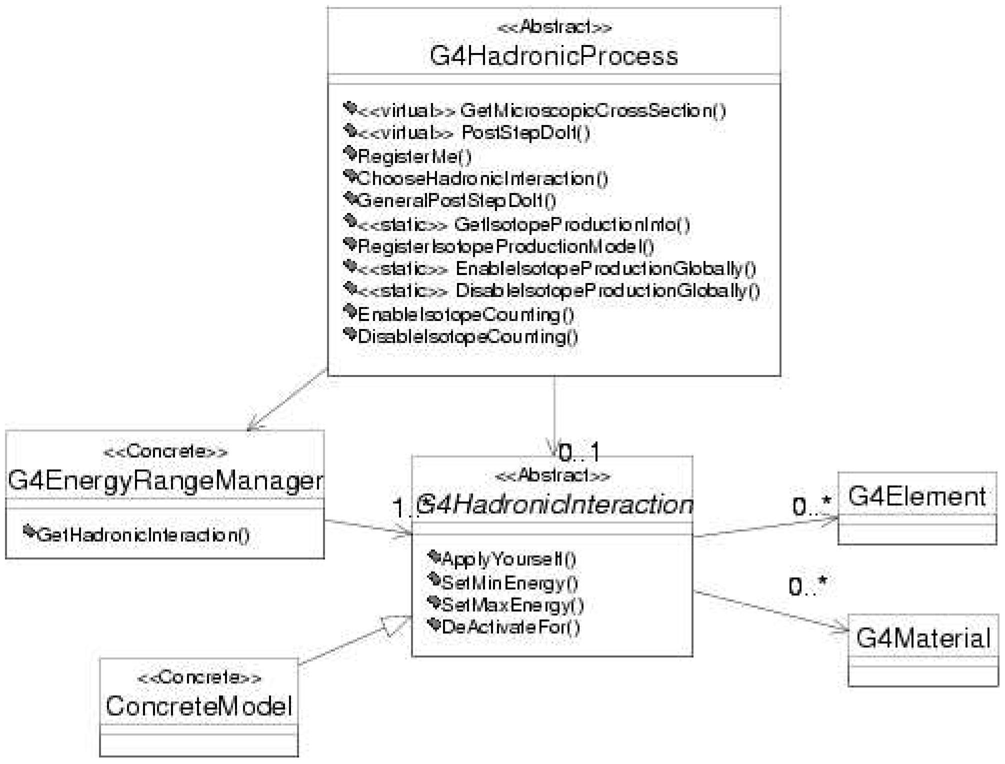


Figure 18.3: Level 2 implementation framework of the hadronic category of GEANT4; final state production aspect.

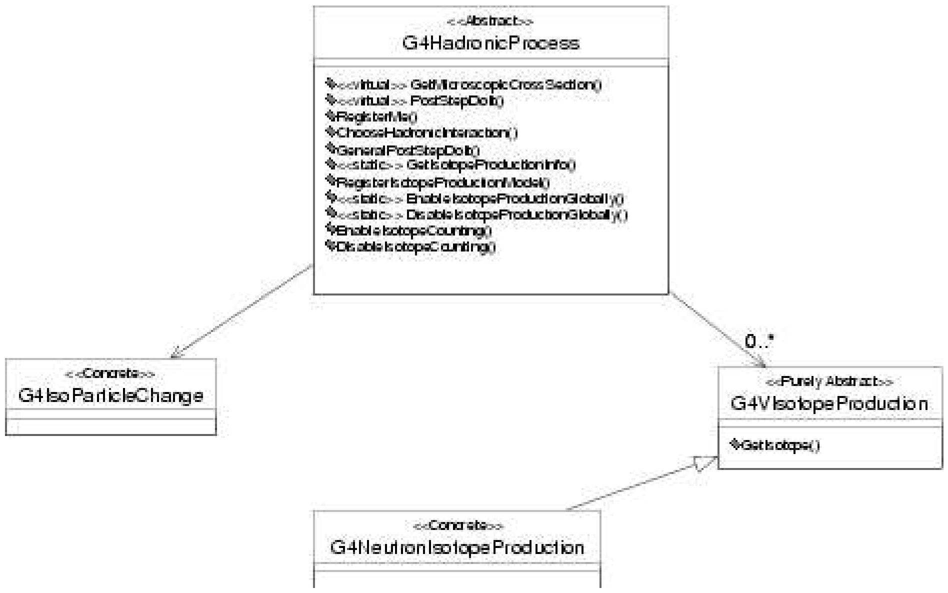

<<Abstract>>
G4HadronicProcess
<<virtual>> GetMicroscopicCrossSection()
<<virtual>> PostStepDoIt()
RegisterMe()
ChooseHadronicInteraction()
GeneralPostStepDoIt()
<<static>> GetIsotopeProductionInfo()
RegisterIsotopeProductionModel()
<<static>> EnableIsotopeProductionGlobally()
<<static>> DisableIsotopeProductionGlobally()
EnableIsotopeCounting()
DisableIsotopeCounting()
<<Concrete>>
G4IsoParticleChange
0..*
<<Purely Abstract>>
G4VIsotopeProduction
GetIsotope()
<<Concrete>>
G4NeutronIsotopeProduction

Figure 18.4: Level 2 implementation framework of the hadronic category of GEANT4; isotope production aspect

senting an object that encapsulates methods and data for calculating total cross sections for a given process in a certain range of validity. The implementations may take any form. It can be a simple equation as well as sophisticated parameterisations, or evaluated data. All cross section data set classes are derived from the abstract class `G4VCrossSectionDataSet`, which declares methods that allow the process inquire, about the applicability of an individual data-set through
`IsApplicable(const G4DynamicParticle*, const G4Element*)`,
and to delegate the calculation of the actual cross-section value through
`GetCrossSection(const G4DynamicParticle*, const G4Element*)`.

Final state production The `G4HadronicInteraction` base class is provided for final state generation. It declares a minimal interface of only one pure virtual method:
`G4VParticleChange* ApplyYourself(const G4Track &, G4Nucleus &)`.
`G4HadronicProcess` provides a registry for final state production models inheriting from `G4HadronicInteraction`. Again, final state production model is meant in very general terms. This can be an implementation of a quark gluon string model[4], a sampling code for ENDF/B data formats [5], or a parametrisation describing only neutron elastic scattering off Silicon up to 300 MeV.

Isotope production For isotope production, a base class (G4VIsotope-Production) is provided. It declares a method
`G4IsoResult * GetIsotope(const G4Track &, const G4Nucleus &)`
that calculates and returns the isotope production information. Any concrete isotope production model will inherit from this class, and implement the method. Again, the modeling possibilities are not limited, and the implementation of concrete production models is not restricted in any way. By convention, the GetIsotope method returns NULL, if the model is not applicable for the current projectile target combination.

Framework functionality:

Cross Sections G4HadronicProcess provides registering possibilities for data sets. A default is provided covering all possible conditions to some approximation. The process stores and retrieves the data sets through a data store that acts like a FILO stack (a Chain of Responsibility with a First In Last Out decision strategy). This allows the user to map out the entire parameter space by overlaying cross section data sets to optimise the overall result. Examples are the cross sections for low energy neutron transport. If these are registered last by the user, they will be used whenever low energy neutrons are encountered. In all other conditions the system falls back on the default, or data sets with earlier registration dates. The fact that the registration is done through abstract base classes with no side-effects allows the user to implement and use his own cross sections. Examples are special reaction cross sections of K^0-nuclear interactions that might be used for ϵ/ϵ' analysis at LHC to control the systematic error.

Final state production The G4HadronicProcess class provides a registration service for classes deriving from G4HadronicInteraction, and delegates final state production to the applicable model. G4HadronicInteraction provides the functionality needed to define and enforce the applicability of a particular model. Models inheriting from G4HadronicInteraction can be restricted in applicability in projectile type and energy, and can be activated/deactivated for individual materials and elements. This allows a user to use final state production models in arbitrary combinations, and to write his own models for final state production. The design is a variant of a Chain of Responsibility. An example would be the likely CMS scenario - the combination of low energy neutron transport with a quantum molecular dynamics[6] or chiral invariant phase space decay[7] model in the case of tracker materials and fast parametrised models for calorimeter materials, with user defined modeling of interactions of spallation nucleons with the most abundant tracker and calorimeter materials.

Isotope production The `G4HadronicProcess` by default calculates the isotope production information from the final state given by the transport model. In addition, it provides a registering mechanism for isotope production models that run in parasitic mode to the transport models and inherit from `G4VIsotopeProduction`. The registering mechanism behaves like a FILO stack, again based on Chain of Responsibility. The models will be asked for isotope production information in inverse order of registration. The first model that returns a non-NULL value will be applied. In addition, the `G4HadronicProcess` provides the basic infrastructure for accessing and steering of isotope-production information. It allows to enable and disable the calculation of isotope production information globally, or for individual processes, and to retrieve the isotope production information through the
`G4IsoParticleChange * GetIsotopeProductionInfo()`
method at the end of each step. The `G4HadronicProcess` is a finite state machine that will ensure the `GetIsotopeProductionInfo` returns a non-zero value only at the first call after isotope production occurred. An example of the use of this functionality is the study of activation of a Germanium detector in a high precision, low background experiment.

18.5 Level 3 Framework - Theoretical Models

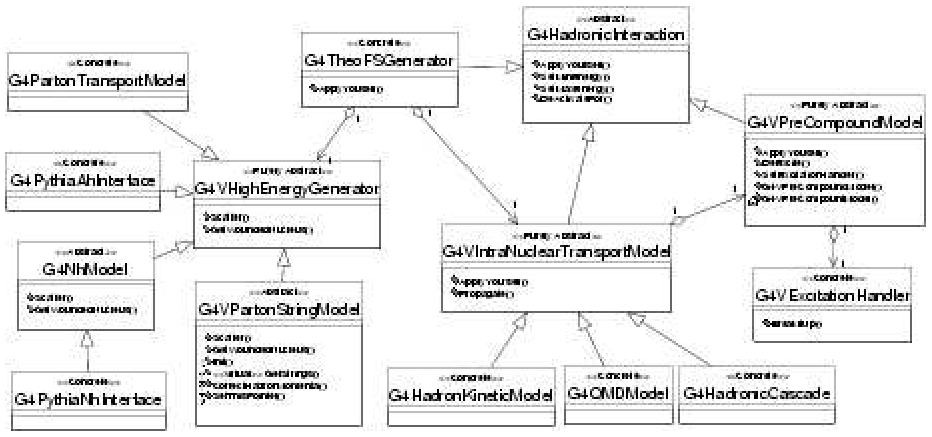


Figure 18.5: Level 3 implementation framework of the hadronic category of GEANT4; theoretical model aspect.

GEANT4 provides at present one implementation framework for theory driven models. The main use-case is that of a user wishing to use theoretical models in general, and to use various intra-nuclear transport or pre-compound models together with models simulating the initial interactions at very high energies.

Requirements

1. Allow to use or adapt any string-parton or parton transport[8] model.
2. Allow to adapt event generators, ex. PYTHIA[9] for final state production in shower simulation.
3. Allow for combination of the above with any intra-nuclear transport (INT).
4. Allow stand-alone use of intra-nuclear transport.
5. Allow for combination of the above with any pre-compound model.
6. Allow stand-alone use of any pre-compound model.
7. Allow for use of any evaporation code.
8. Allow for seamless integration of user defined components for any of the above.

Design and interfaces To provide the above flexibility, the following abstract base classes have been implemented:

- `G4VHighEnergyGenerator`
- `G4VIntranuclearTransportModel`
- `G4VPreCompoundModel`
- `G4VExcitationHandler`

In addition, the class `G4TheoFSGenerator` is provided to orchestrate interactions between these classes. The class diagram is shown in Fig. 18.5.

`G4VHighEnergyGenerator` serves as base class for parton transport or parton string models, and for Adapters to event generators. This class declares two methods, `Scatter`, and `GetWoundedNucleus`.

The base class for INT inherits from `G4HadronicInteraction`, making any concrete implementation usable as a stand-alone model. In doing so, it re-declares the `ApplyYourself` interface of `G4HadronicInteraction`, and adds a second interface, `Propagate`, for further propagation after high energy interactions. `Propagate` takes as arguments a three-dimensional model of a wounded nucleus, and a set of secondaries with energies and positions.

The base class for pre-equilibrium decay models, `G4VPreCompoundModel`, inherits from `G4HadronicInteraction`, again making any concrete implementation usable as stand-alone model. It adds a pure virtual `DeExcite` method for further evolution of the system when intra-nuclear transport assumptions break down. `DeExcite` takes a `G4Fragment`, the GEANT4 representation of an excited nucleus, as argument.

The base class for evaporation phases, `G4VExcitationHandler`, declares an abstract method, `BreakItUP()`, for compound decay.

Framework functionality The `G4TheoFSGenerator` class inherits from `G4HadronicInteraction`, and hence can be registered as a model for final state production with a hadronic process. It allows a concrete implementation of `G4VIntranuclearTransportModel` and `G4VHighEnergyGenerator` to be registered, and delegates initial interactions, and intra-nuclear transport of the corresponding secondaries to the respective classes. The design is a complex variant of a Strategy. The most spectacular application of this pattern is the use of parton-string models for string excitation, quark molecular dynamics for correlated string decay, and quantum molecular dynamics for transport, a combination which promises to result in a coherent description of quark gluon plasma in high energy nucleus-nucleus interactions.

The class `G4VIntranuclearTransportModel` provides registering mechanisms for concrete implementations of `G4VPreCompoundModel`, and provides concrete intra-nuclear transports with the possibility of delegating pre-compound decay to these models.

`G4VPreCompoundModel` provides a registering mechanism for compound decay through the `G4VExcitationHandler` interface, and provides concrete implementations with the possibility of delegating the decay of a compound nucleus.

The concrete scenario of `G4TheoFSGenerator` using a dual parton model and a classical cascade, which in turn uses an exciton pre-compound model that delegates to an evaporation phase, would be the following: `G4TheoFS-Generator` receives the conditions of the interaction; a primary particle and a nucleus. It asks the dual parton model to perform the initial scatterings, and return the final state, along with the by then damaged nucleus. The nucleus records all information on the damage sustained. `G4TheoFSGenerator` forwards all information to the classical cascade, that propagates the particles in the already damaged nucleus, keeping track of interactions, further damage to the nucleus, etc.. Once the cascade assumptions break down, the cascade will collect the information of the current state of the hadronic system, like excitation energy and number of excited particles, and interpret it as a pre-compound system. It delegates the decay of this to the exciton model. The exciton model will take the information provided, and calculate transitions and emissions, until the number of excitons in the system equals the mean number of excitons expected in equilibrium for the current excitation energy. Then it hands over to the evaporation phase. The evaporation phase decays the residual nucleus, and the Chain of Command rolls back to `G4TheoFSGenerator`, accumulating the information produced in the various levels of delegation.

18.6 Level 4 Frameworks - String Parton Models and Intra-Nuclear Cascade

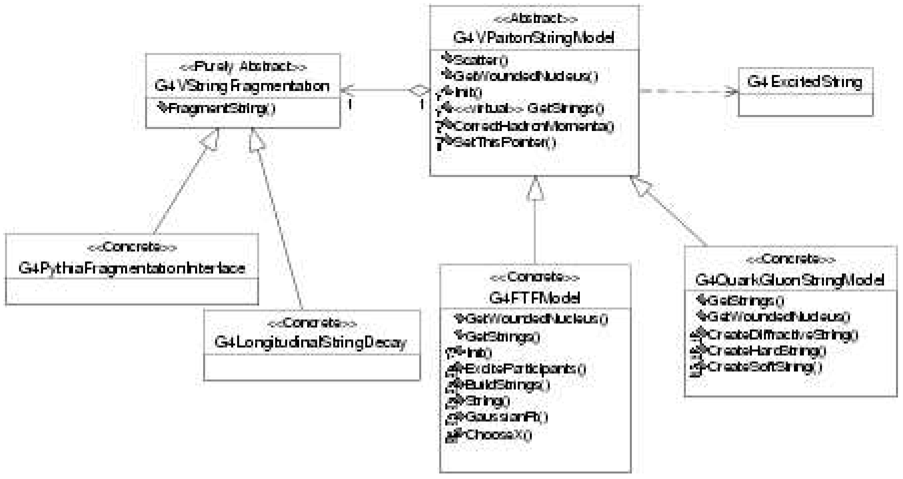


Figure 18.6: Level 4 implementation framework of the hadronic category of GEANT4; parton string aspect.

The use-cases of this level are related to commonalities and detailed choices in string-parton models and cascade models. They are use-cases of an expert user wishing to alter details of a model, or a theoretical physicist, wishing to study details of a particular model.

Requirements

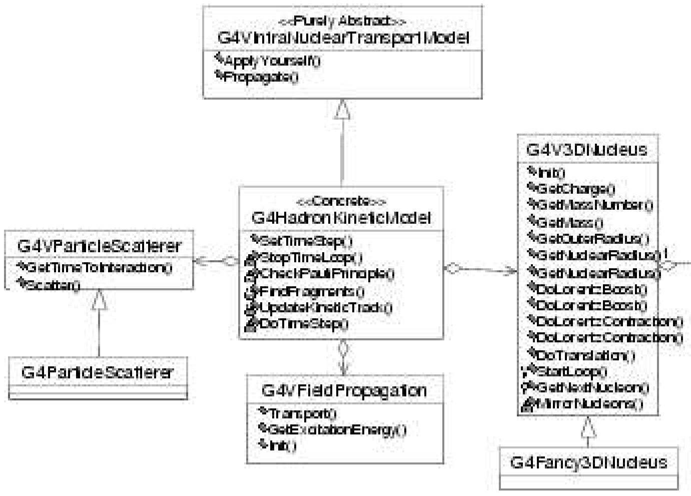


Figure 18.7: Level 4 implementation framework of the hadronic category of GEANT4; intra-nuclear transport aspect.

1. Allow to select string decay algorithm
2. Allow to select string excitation.
3. Allow the selection of concrete implementations of three-dimensional models of the nucleus
4. Allow the selection of concrete implementations of final state and cross sections in intra-nuclear scattering.

Design and interfaces To fulfil the requirements on string models, two abstract classes are provided, the `G4VPartonStringModel` and the `G4VStringFragmentation`. The base class for parton string models, `G4VPartonStringModel`, declares the `GetStrings()` pure virtual method. `G4VStringFragmentation`, the pure abstract base class for string fragmentation, declares the interface for string fragmentation.

To fulfill the requirements on intra-nuclear transport, two abstract classes are provided, `G4V3DNucleus`, and `G4VScatterer`. At this point in time, the usage of these intra-nuclear transport related classes by concrete codes is not enforced by designs, as the details of the cascade loop are still model dependent, and more experience has to be gathered to achieve standardisation. It is within the responsibility of the implementers of concrete intra-nuclear transport codes to use the abstract interfaces as defined in these classes.

The class diagram is shown in figure 18.6 for the string parton model aspects, and in figure 18.7 for the intra-nuclear transport.

Framework functionality Again variants of Strategy, Bridge and Chain of Responsibility are used. `G4VPartonStringModel` implements the initialisation of a three-dimensional model of a nucleus, and the logic of scattering. It delegates secondary production to string fragmentation through a `G4VStringFragmentation` pointer. It provides a registering service for the concrete string fragmentation, and delegates the string excitation to derived classes. Selection of string excitation is through selection of derived class. Selection of string fragmentation is through registration.

18.7 Level 5 Framework - String De-excitation

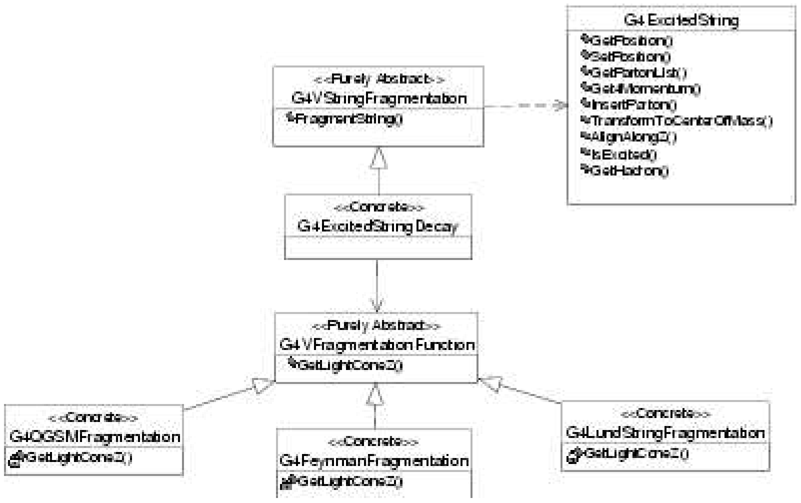


Figure 18.8: Level 5 implementation framework of the hadronic category of GEANT4; string fragmentation aspect.

The use-case of this level is that of a user or theoretical physicist wishing to understand the systematic effects involved in combining various fragmentation functions with individual types of string fragmentation. Note that this framework level is meeting the current state of the art, making extensions and changes of interfaces in subsequent releases likely.

Requirements

1. Allow the selection of fragmentation function.

Design and interfaces A base class for fragmentation functions, `G4VFragmentation-Function`, is provided. It declares the `GetLightConeZ()` interface.

Framework functionality The design is a basic Strategy. The class diagram is shown in Fig. 18.8. At this point in time, the usage of the `G4VFragmentationFunction` is not enforced by design, but made available from the `G4VStringFragmentation` to an implementer of a concrete string decay. `G4VStringFragmentation` provides a registering mechanism for the concrete fragmentation function. It delegates the calculation of z_f of the hadron to split of the string to the concrete implementation. Standardisation in this area is expected.

Bibliography

[1] The GEANT 4 Collaboration, CERN/DRDC/94–29, DRDC/P58 1994.

[2] E. Gamm et al., Design Patterns, Addison-Wesley Professional Computing Series, 1995.

[3] http://cern.ch/wwwasd/geant4/G4UsersDocuments/Overview/html/index.html

[4] Kaidalov A. B., Ter-Martirosyan K. A., Phys. Lett. **B117** 247 (1982);

[5] Data Formats and Procedures for the Evaluated Nuclear Data File, National Nuclear Data Center, Brookhave National Laboratory, Upton, NY, USA.

[6] For example: VUU and (R)QMD model of high-energy heavy ion collisions. H. Stocker et al., Nucl. Phys. A538, 53c-64c (1992)

[7] P.V. Degtyarenko, M.V. Kossov, H.P. Wellisch, Eur. Phys J. A 8, 217-222 (2000)

[8] VNI 3.1, Klaus Geiger (Brookhaven), BNL-63762, Comput. Phys. Commun. 104, 70-160 (1997)

[9] M. Bertini, L. Lönnblad, T. Sjörstrand, Pythia version 7-0.0 – a proof-of-concept version, LU-TP 00-23, hep-ph/0006152, May 2000

Chapter 19

Visualization

The following sections contain various information for extending class functionalities of GEANT4 visualization:

- User's Guide for Application Developers, Chapter 8 - Visualization
- User's Guide for Toolkit Developers, Chapter 3 - Object-oriented Analysis and Design of GEANT4 Classes, Section 9 - Visualization

Part IV

Appendix

Chapter 20

Class Diagrams for the Geometry Category

Figure 20.1 shows the OO design of the logical volume. Figure 20.2 shows the OO design of the physical volume. Figure 20.3 shows the OO design of the CSG solids. Figure 20.4 shows the OO design of the boolean solids. Figure 20.5 shows the OO design of the specific solids. Figure 20.6 shows the OO design of the BREP solids. Figure 20.7 shows the OO design of the BREP curves. Figure 20.8 shows the OO design of the BREP surfaces. Figure 20.9 shows the OO design for reflections of solids. Figure 20.10 shows the OO design of the touchables. Figure 20.11 shows the OO design of reference counting of touchables. Figure 20.12 shows the OO design of smart voxels. Figure 20.13 shows the OO design of the navigator. Figure 20.14 shows the OO design of detector regions.

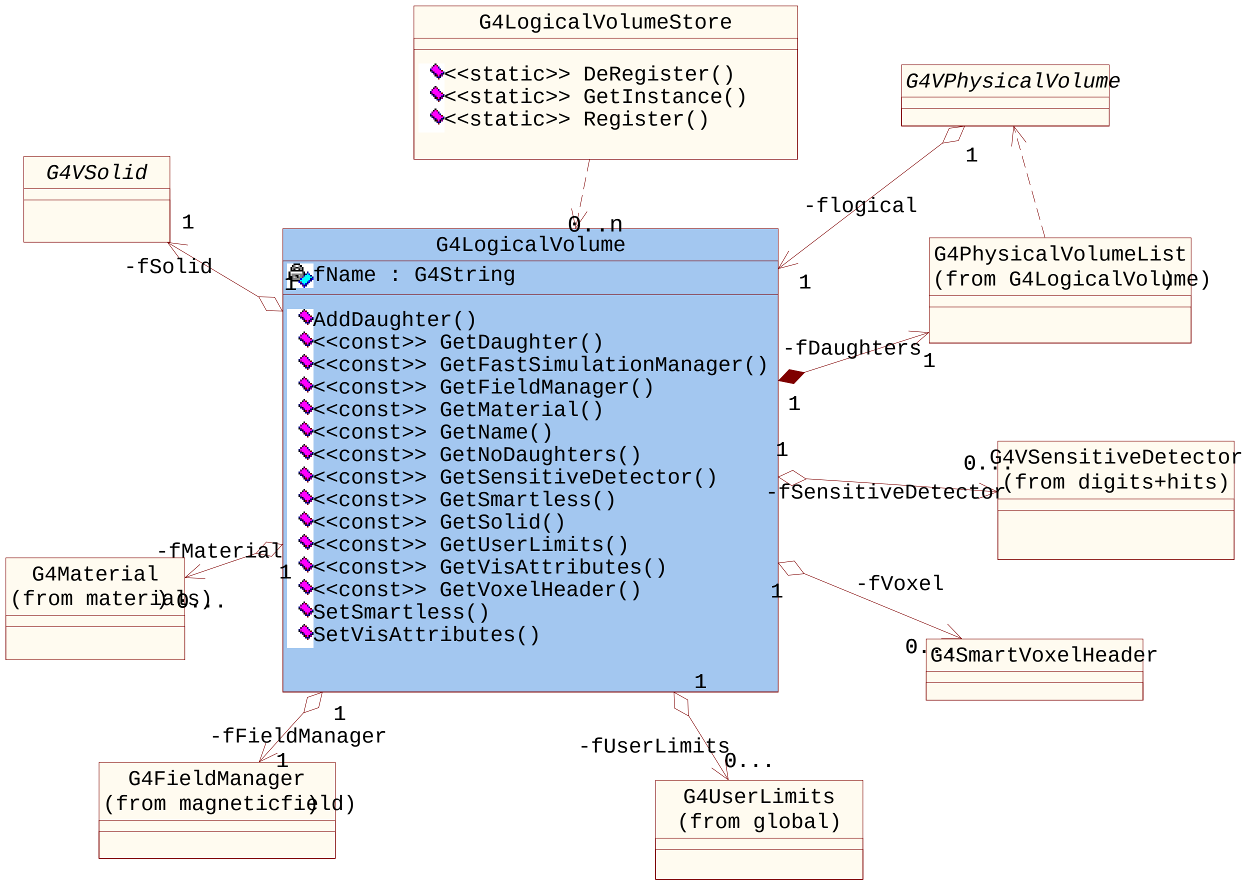


Figure 20.1: Logical volumes

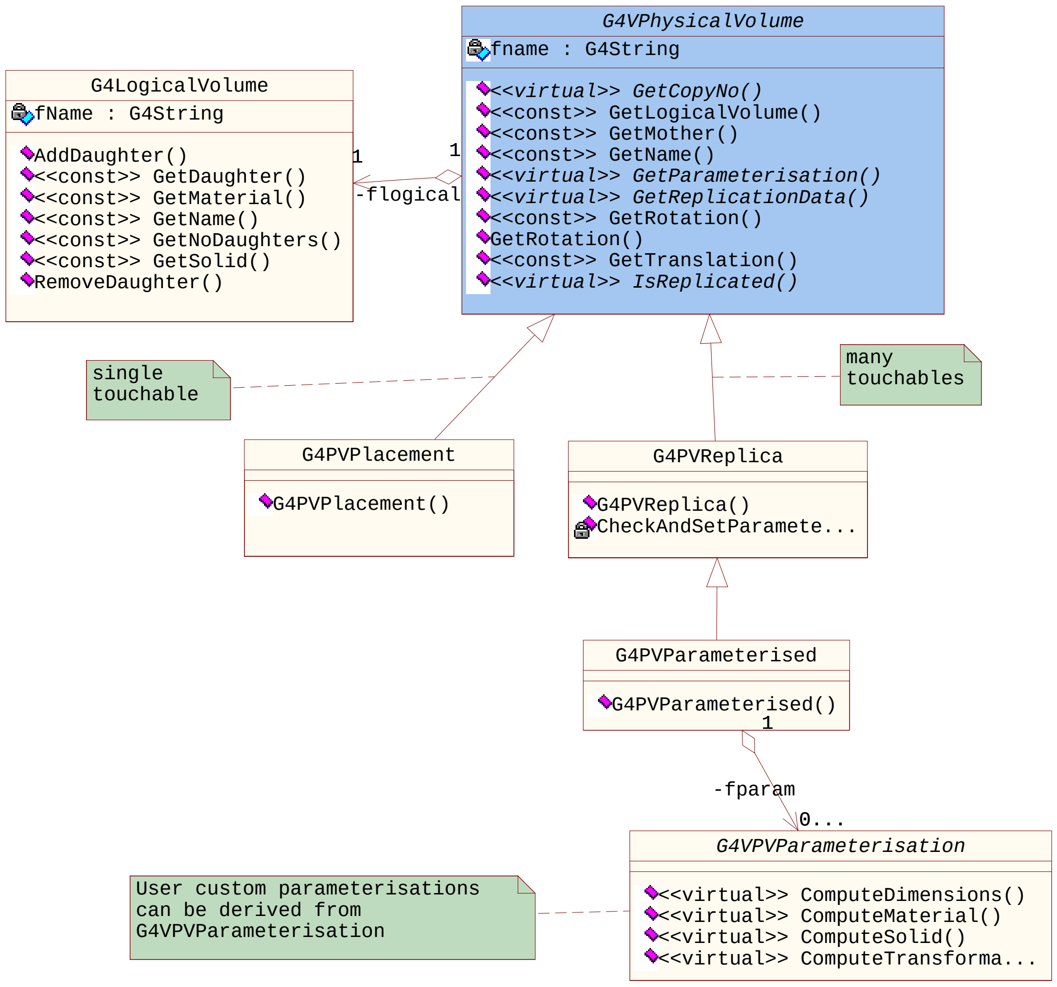
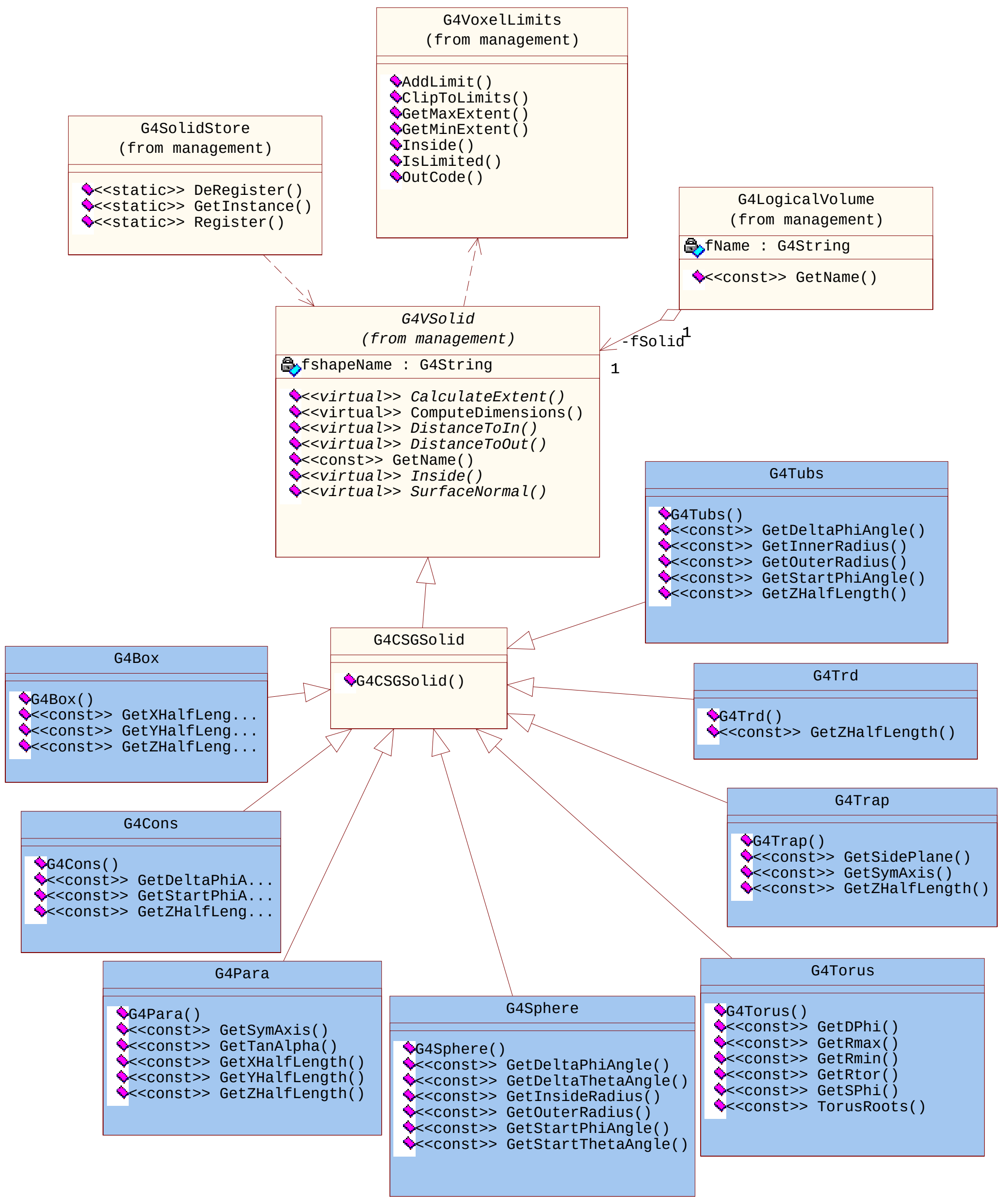


Figure 20.2: Physical volumes



G4VoxelLimits
(from management)
AddLimit()
ClipToLimits()
GetMaxExtent()
GetMinExtent()
Inside()
IsLimited()
OutCode()
G4SolidStore
(from management)
<<static>> DeRegister()
<<static>> GetInstance()
<<static>> Register()
G4LogicalVolume
(from management)
fName : G4String
<<const>> GetName()
-fSolid
1
1
G4VSolid
(from management)
fshapeName : G4String
<<virtual>> CalculateExtent()
<<virtual>> ComputeDimensions()
<<virtual>> DistanceToIn()
<<virtual>> DistanceToOut()
<<const>> GetName()
<<virtual>> Inside()
<<virtual>> SurfaceNormal()
G4Tubs
G4Tubs()
<<const>> GetDeltaPhiAngle()
<<const>> GetInnerRadius()
<<const>> GetOuterRadius()
<<const>> GetStartPhiAngle()
<<const>> GetZHalfLength()
G4CSGSolid
G4CSGSolid()
G4Box
G4Box()
<<const>> GetXHalfLeng...
<<const>> GetYHalfLeng...
<<const>> GetZHalfLeng...
G4Trd
G4Trd()
<<const>> GetZHalfLength()
G4Trap
G4Trap()
<<const>> GetSidePlane()
<<const>> GetSymAxis()
<<const>> GetZHalfLength()
G4Cons
G4Cons()
<<const>> GetDeltaPhiA...
<<const>> GetStartPhiA...
<<const>> GetZHalfLeng...
G4Para
G4Para()
<<const>> GetSymAxis()
<<const>> GetTanAlpha()
<<const>> GetXHalfLength()
<<const>> GetYHalfLength()
<<const>> GetZHalfLength()
G4Sphere
G4Sphere()
<<const>> GetDeltaPhiAngle()
<<const>> GetDeltaThetaAngle()
<<const>> GetInsideRadius()
<<const>> GetOuterRadius()
<<const>> GetStartPhiAngle()
<<const>> GetStartThetaAngle()
G4Torus
G4Torus()
<<const>> GetDPhi()
<<const>> GetRmax()
<<const>> GetRmin()
<<const>> GetRtor()
<<const>> GetSPhi()
<<const>> TorusRoots()

Figure 20.3: CSG solids

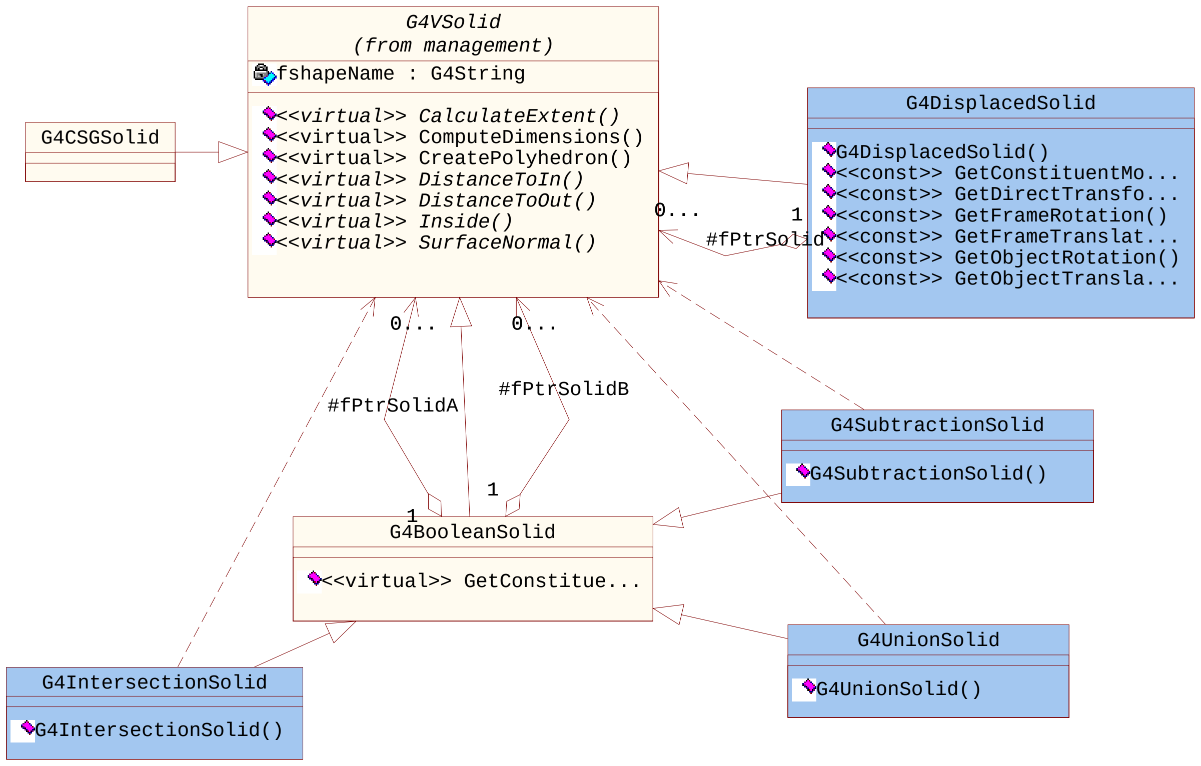

G4VSolid
(from management)
fshapeName : G4String
<<virtual>> CalculateExtent()
<<virtual>> ComputeDimensions()
<<virtual>> CreatePolyhedron()
<<virtual>> DistanceToIn()
<<virtual>> DistanceToOut()
<<virtual>> Inside()
<<virtual>> SurfaceNormal()
G4CSGSolid
G4DisplacedSolid
G4DisplacedSolid()
<<const>> GetConstituentMo...
<<const>> GetDirectTransfo...
<<const>> GetFrameRotation()
<<const>> GetFrameTranslat...
<<const>> GetObjectRotation()
<<const>> GetObjectTransla...
0...
1
#fPtrSolid
0...
0...
#fPtrSolidA
#fPtrSolidB
1
1
G4SubtractionSolid
G4SubtractionSolid()
G4BooleanSolid
<<virtual>> GetConstitue...
G4UnionSolid
G4UnionSolid()
G4IntersectionSolid
G4IntersectionSolid()

Figure 20.4: Boolean solids

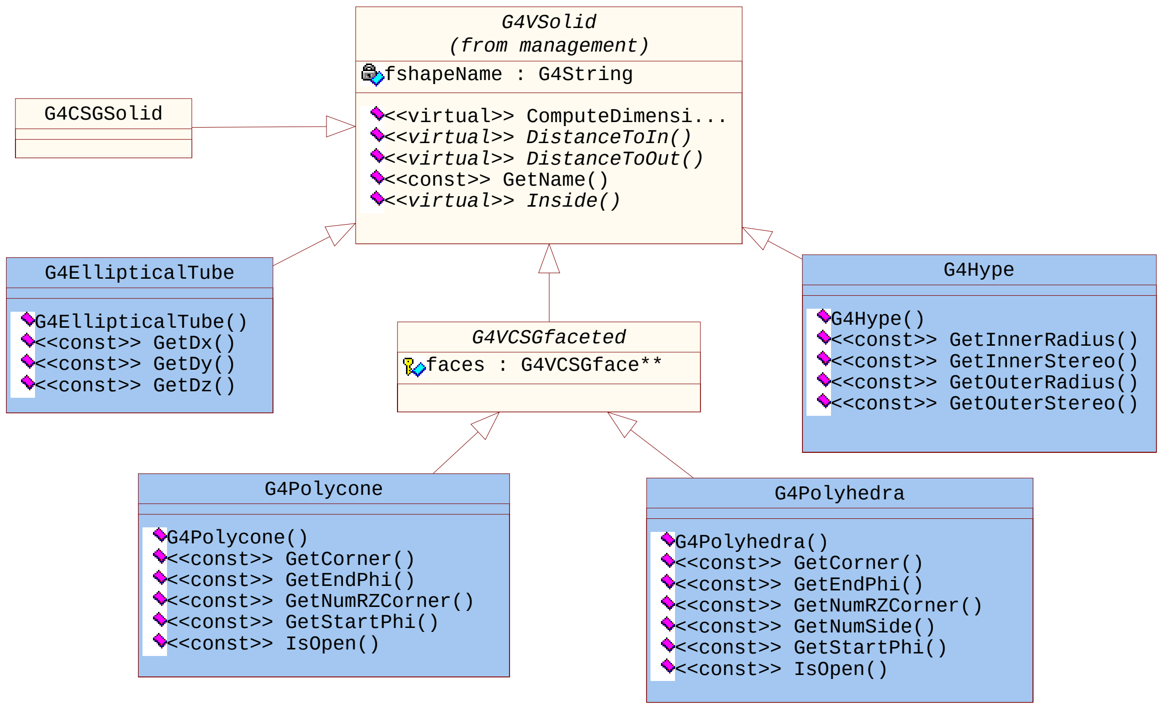


Figure 20.5: Specific solids

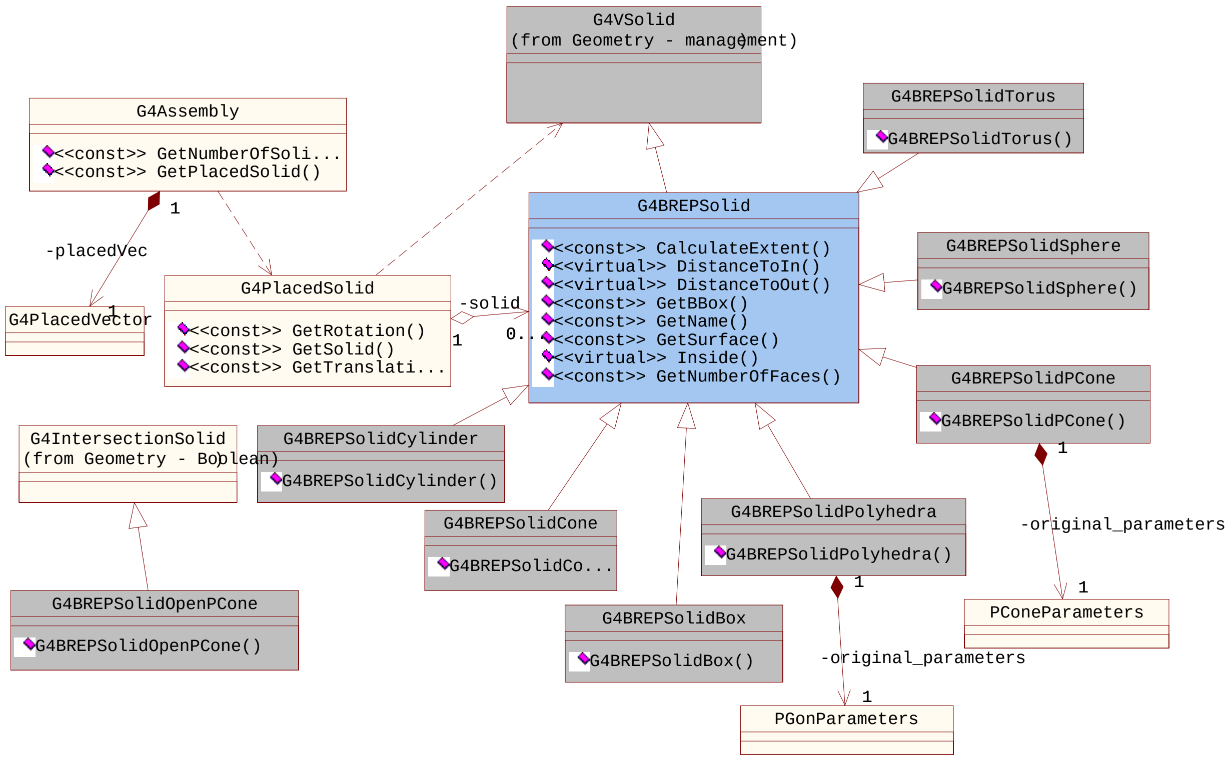


Figure 20.6: BREP solids

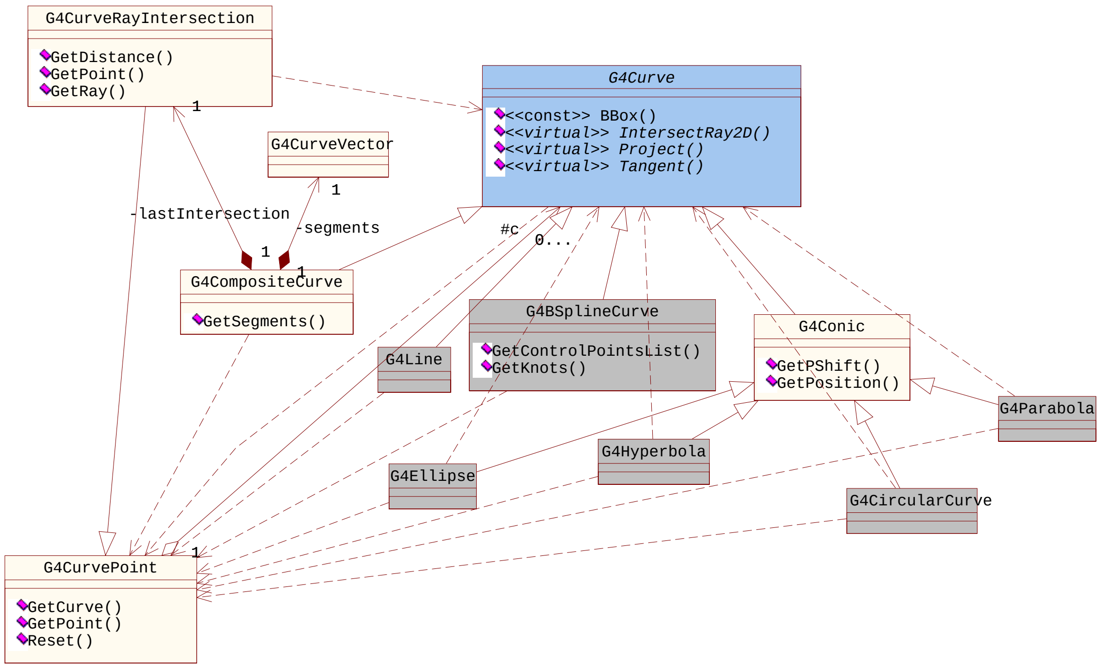


Figure 20.7: BREP curves

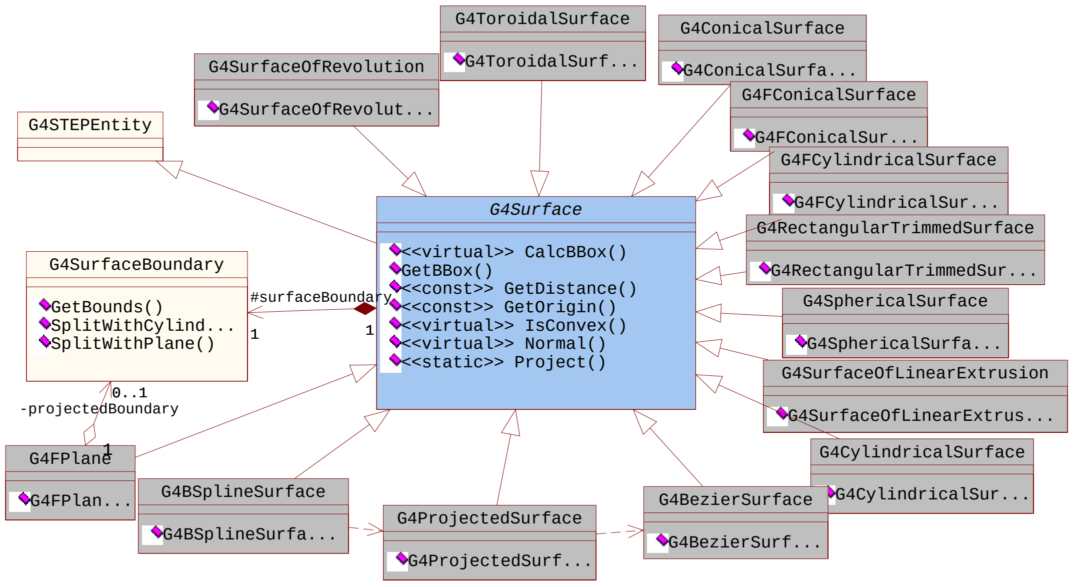


Figure 20.8: BREP surfaces

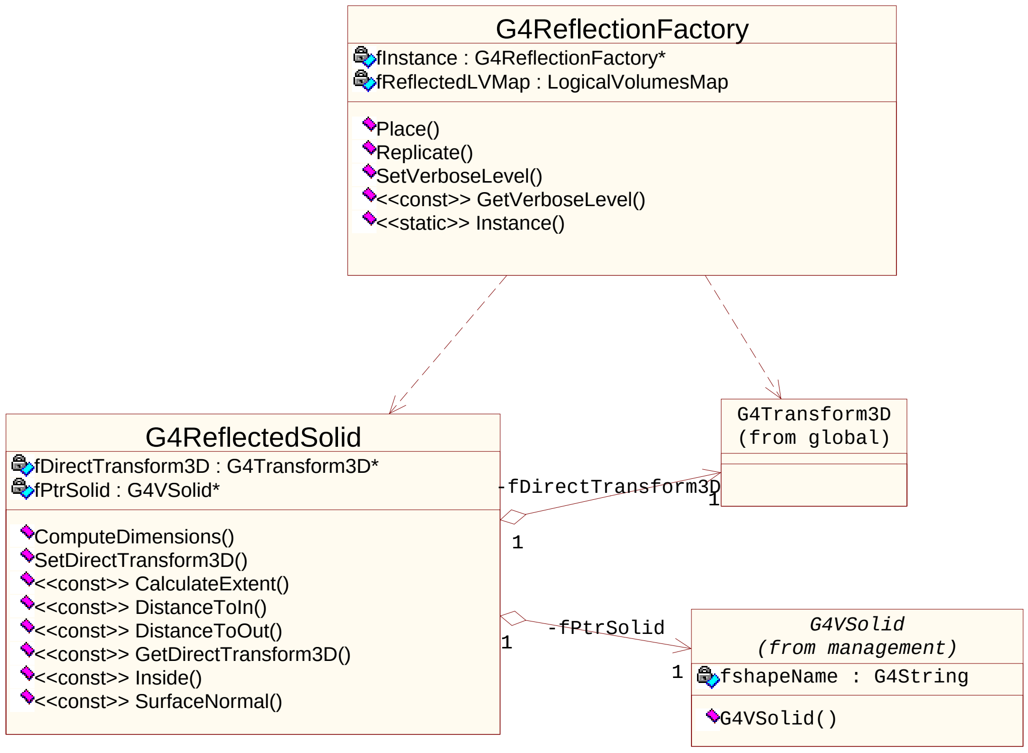


Figure 20.9: Reflections of solids

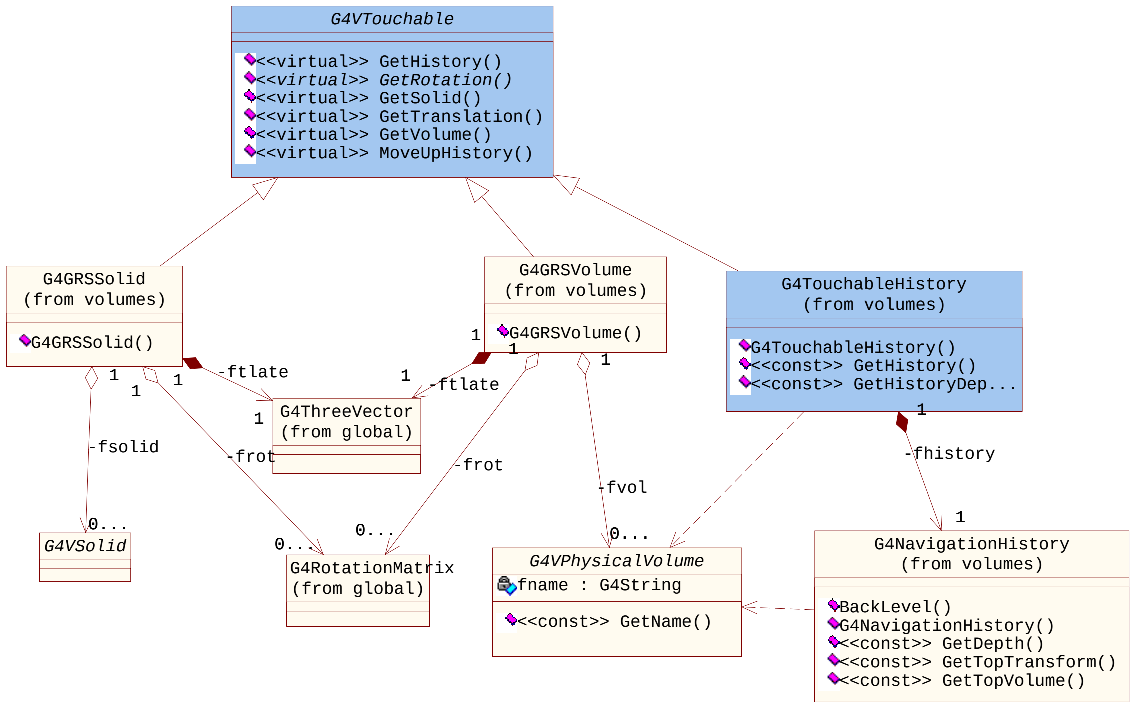


Figure 20.10: Touchables

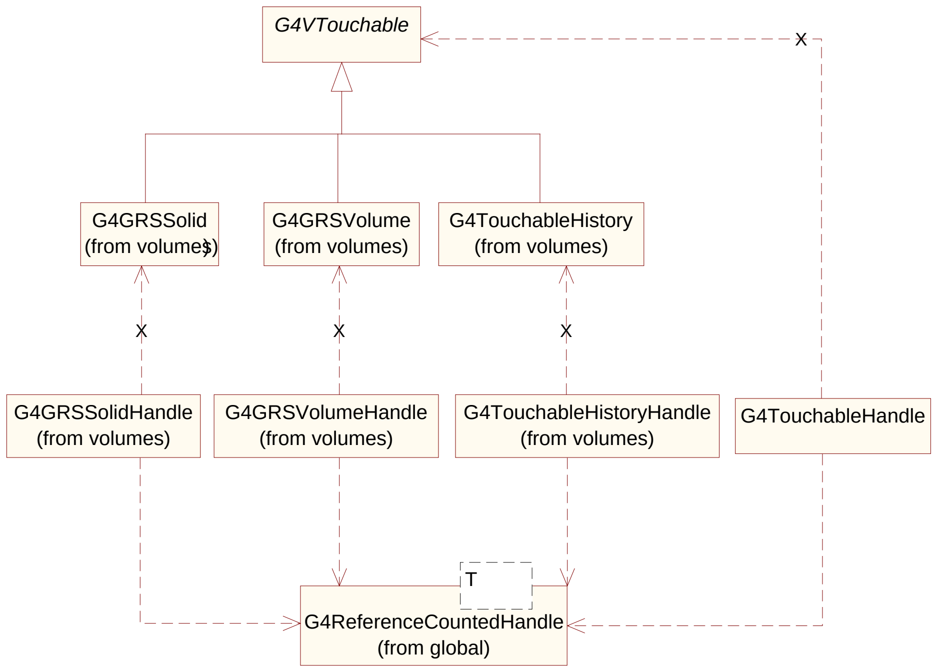


Figure 20.11: Reference counting for touchables

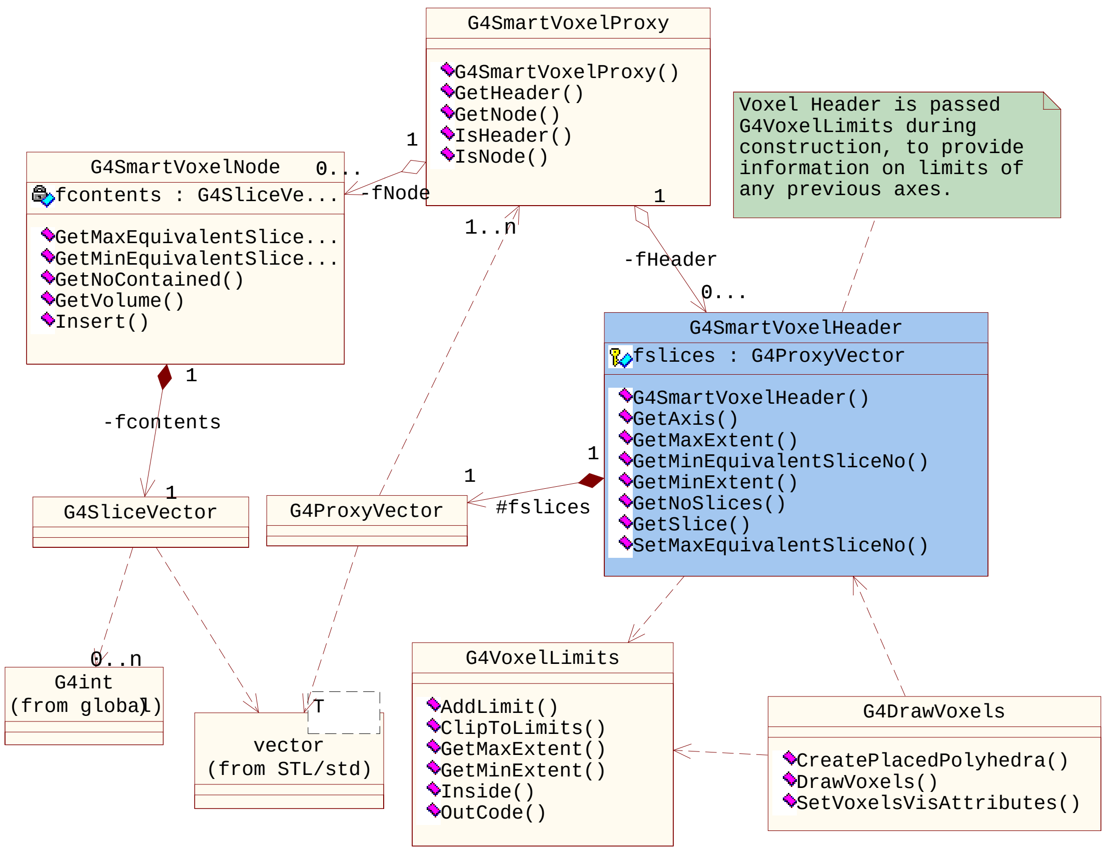
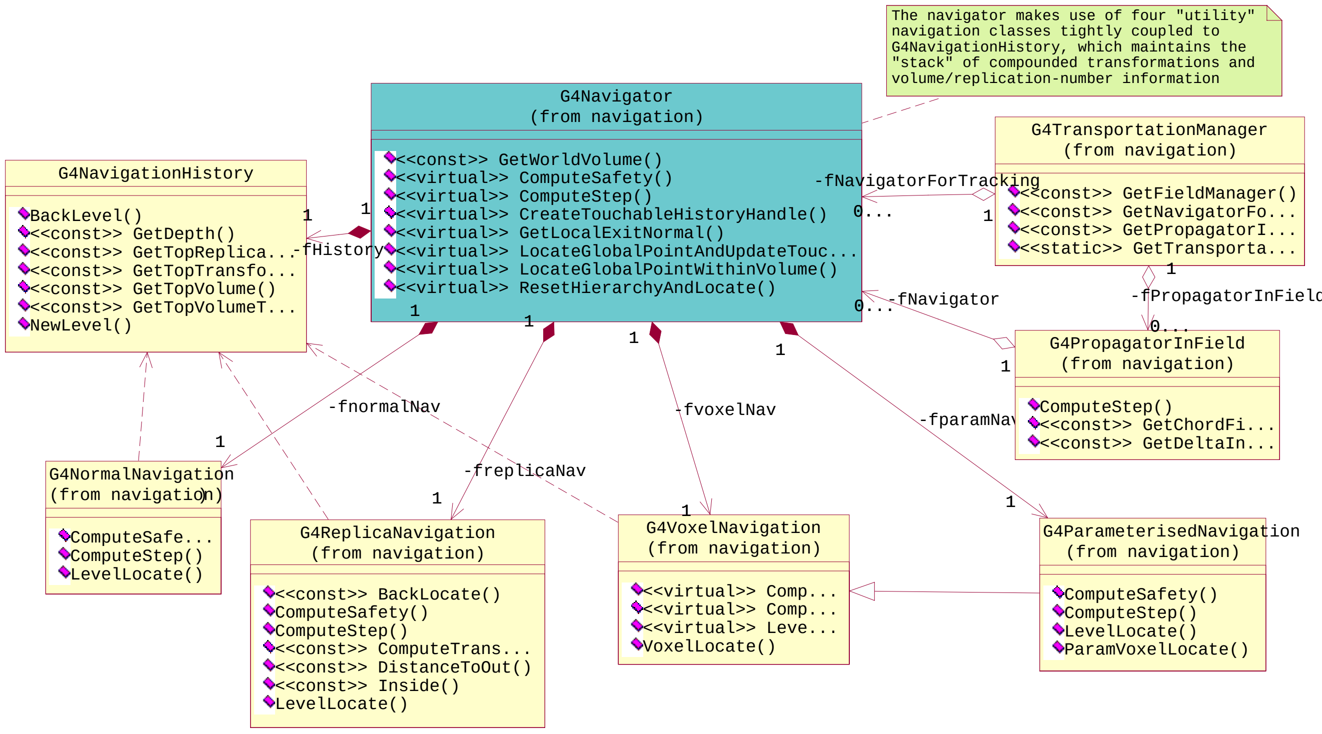


Figure 20.12: Smart Voxels



The navigator makes use of four "utility" navigation classes tightly coupled to G4NavigationHistory, which maintains the "stack" of compounded transformations and volume/replication-number information
G4Navigator
(from navigation)
<<const>> GetWorldVolume()
<<virtual>> ComputeSafety()
<<virtual>> ComputeStep()
<<virtual>> CreateTouchableHistoryHandle()
<<virtual>> GetLocalExitNormal()
<<virtual>> LocateGlobalPointAndUpdateTouc...
<<virtual>> LocateGlobalPointWithinVolume()
<<virtual>> ResetHierarchyAndLocate()
G4NavigationHistory
BackLevel()
<<const>> GetDepth()
<<const>> GetTopReplica...
<<const>> GetTopTransfo...
<<const>> GetTopVolume()
<<const>> GetTopVolumeT...
NewLevel()
-fHistory
G4TransportationManager
(from navigation)
<<const>> GetFieldManager()
<<const>> GetNavigatorFo...
<<const>> GetPropagatorI...
<<static>> GetTransporta...
-fNavigatorForTracking
-fNavigator
-fPropagatorInField
G4PropagatorInField
(from navigation)
ComputeStep()
<<const>> GetChordFi...
<<const>> GetDeltaIn...
-fnormalNav
-freplicaNav
-fvoxelNav
-fparamNav
G4NormalNavigation
(from navigation)
ComputeSafe...
ComputeStep()
LevelLocate()
G4ReplicaNavigation
(from navigation)
<<const>> BackLocate()
ComputeSafety()
ComputeStep()
<<const>> ComputeTrans...
<<const>> DistanceToOut()
<<const>> Inside()
LevelLocate()
G4VoxelNavigation
(from navigation)
<<virtual>> Comp...
<<virtual>> Comp...
<<virtual>> Leve...
VoxelLocate()
G4ParameterisedNavigation
(from navigation)
ComputeSafety()
ComputeStep()
LevelLocate()
ParamVoxelLocate()
1
0...

Figure 20.13: Navigator

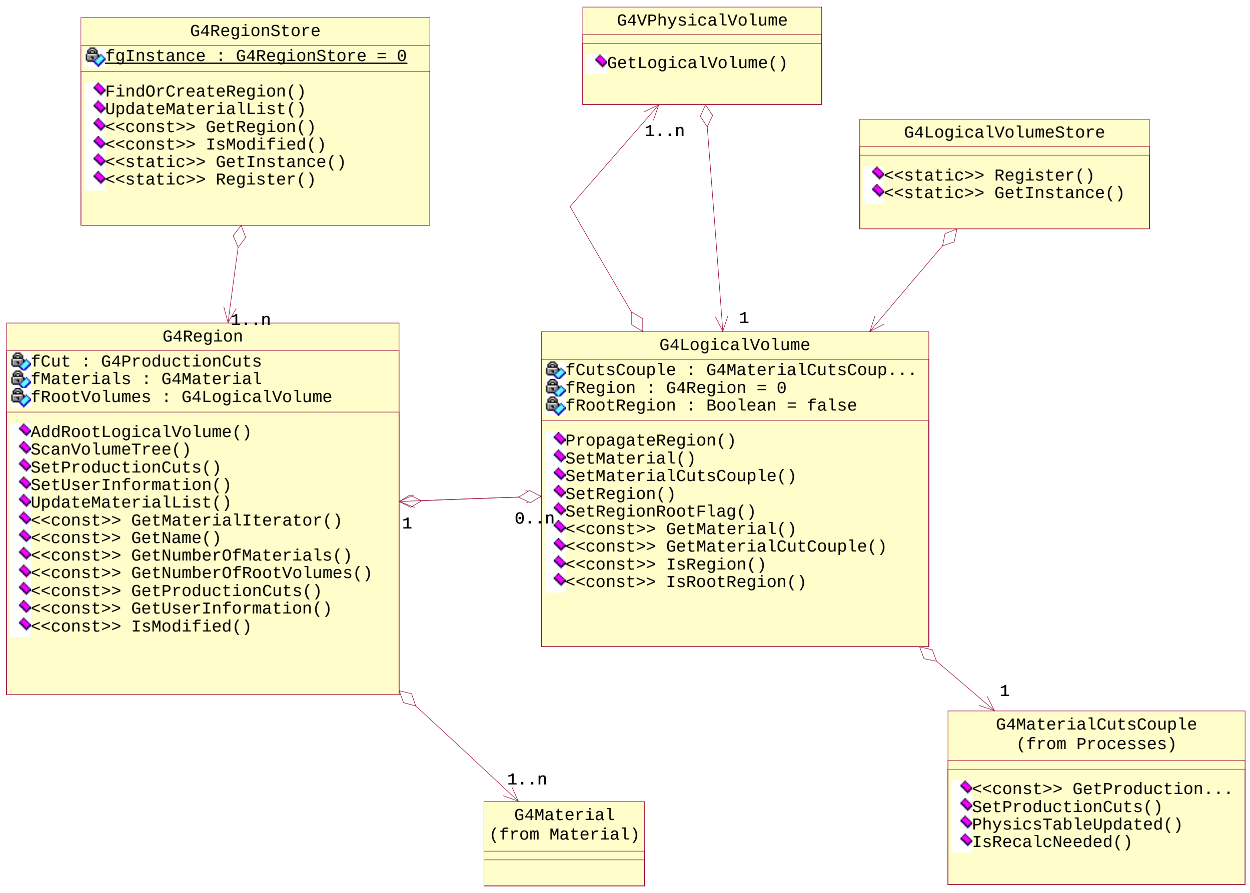


Figure 20.14: Regions